Spring 1-1-2011

Invalidating Transactions: Optimizations, Theory, Guarantees, and Unification

Justin E. Gottschlich
University of Colorado at Boulder, justin@nodeka.com

Invalidating Transactions: Optimizations, Theory, Guarantees, and Unification

by

Justin E. Gottschlich

B.S., Colorado State University, 2002

M.S., University of Colorado-Boulder, 2007

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer, and Energy Engineering

2011

This thesis entitled:
Invalidating Transactions: Optimizations, Theory, Guarantees, and Unification
written by Justin E. Gottschlich
has been approved for the Department of Electrical, Computer, and Energy Engineering

Jeremy G. Siek, Ph.D. (Chair)

Manish Vachharajani, Ph.D. (Co-Chair)

Aaron Bradley, Ph.D.

Fabio Somenzi, Ph.D.

Maurice Herlihy, Ph.D. (Outside Member)

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Gottschlich, Justin E. (Ph.D., Electrical, Computer, and Energy Engineering)

Invalidating Transactions: Optimizations, Theory, Guarantees, and Unification

Thesis directed by Professor Jeremy G. Siek, Ph.D. (Chair)

Abstract

Transactional memory (TM) is a modern concurrency control paradigm that reduces the difficulty of parallel programming. TM also reduces some unnecessary program serialization by allowing operations from different critical sections, called transactions, to execute concurrently. Although allowing transactions to execute concurrently can increase throughput, care is needed to avoid memory access conflicts between transactions that can lead to incorrect program states.

To prevent such incorrect program states, TM systems identify conflicts between transactions before such illegal states become part of the visible program state. To do this, when two or more transactions conflict, the TM stalls or rolls back some number of transactions to ensure the program state remains *serializable*, the main correctness criterion for TM systems. The process of identifying when transactions conflict is called *conflict detection* and is a significant source of overhead.

To improve the performance of TM, researchers have found optimizations that reduce the cost of conflict detection. However, many of these TMs perform one aspect of conflict detection in the same manner. That is, they perform *commit-time validation*, where a transaction is analyzed for conflicts with previously committed transactions during its commit phase. While commit-time validation has certain benefits, it also has drawbacks that make it a suboptimal conflict detection strategy for certain environments.

In this work we present a conflict detection strategy called *full invalidation* where the TM resolves all conflicts between a given transaction and all other active transactions before the given transaction commits. Full invalidation has a number of advantages over validation such as improved performance, enforceable execution guarantees, reduced conflict speculation, reduced conflict anal-

ysis space and time overhead, and simplified integration of optimistic and pessimistic concurrency control.

We analyze full invalidation in the following ways. First, we compare and contrast InvalSTM, a software transactional memory (STM) that implements full invalidation, against TL2, a state-of-the-art STM that uses commit-time validation. Next we present a new theoretical model for TM systems and use the model to prove that histories accepted by a full invalidation system are both conflict and view serializable. We then demonstrate that a full invalidation STM is notably more efficient (by upwards of 100×) than a commit-time validation STM for programs where transactional priority must be respected. Last, we show that a full invalidation STM can reduce the TM implementation complexity and the TM operational overhead when using optimistic (transactions) and pessimistic (locks) critical sections in the same program.

Dedication

To Lori, my wife and best friend.

Acknowledgements

First and foremost, I would like to thank my advisor Professor Jeremy G. Siek and my co-advisor Professor Manish Vachharajani. Without your patience, guidance, support, and allowance of freedom, this work would not exist. Each journey through a Ph.D. is different and my experience with Professors Siek and Vachharajani has been nothing short of amazing. I have been blessed with two advisors that have been there to challenge me and encourage me at exactly the right times. Perhaps most importantly, Professors Siek and Vachharajani have provided me with the insight to understand and appreciate the many important aspects of first-class research. For that, I will always be in your debt.

I would also like to thank my committee members Professors Aaron Bradley, Fabio Somenzi, and Maurice Herlihy. Professor Bradley you have been an inspiration to me as I see in you someone who has accomplished so much so early in their career. Your presence on my committee has helped press through the theoretical rigor that my research needed and you have taught me that precision is very important. Professor Somenzi I would like to thank you for being so supportive of my research and encouraging me to push through the completion of my doctorate so I could continue doing what I love: research. Your encouragement to me and to the rest of my committee has been unquestionably powerful. Finally, I would like to thank Professor Herlihy. It is a great honor to have you be part of my doctorate work as you and Eliot Moss invented the field that this dissertation is about. Professor Herlihy, you have taken me under your wing, even though I have been so far away, and you have mentored me countless times in subtle but extremely important areas. I am truly fortunate to consider you as one of my mentors and friends.

I would not be where I am today without the ongoing support and love of my family. I would like to thank my parents, Dorothy and Chris Bell and Ron and Nora Gottschlich. You have cultivated me into who I am today and I am forever grateful. To my brother Paul, thank you for teaching me about logic, math, and programming, and helping me realize how exciting these areas can be. To my big, little brother Thomas, thank you for reminding me how to have fun and laugh. To my sister Sarah, thank you for being a wonderful person and one of the hardest working people I know. To my sister Erin, thank you for inspiring me to read more by your consistent example. To my sister Devi, wherever you are, thank you for growing up with me and being there for me when we were so young. To my sister Krystina, you are an amazing addition to our family, thank you for being the incredible mother and teaching all of us how to be more generous. In addition to my family, I would like to thank my family by marriage: Bud and Cathy Peek, Brad and Heather Peek, Matt and Gina Peek, Zach and Laura Peek, and all my wonderful aunts, uncles, cousins, nieces, and nephews. I could not have imagined a better, more amazing, set of in-laws. You are wonderful people. My dogs Max and Chase also need to be thanked. You two have reminded me that it is okay to take a break. You have also reminded me (and demonstrated to me, personally) that we tend to have the most energy right after a nice midday nap!

I would also like to thank my colleagues and fellow researchers who have been supporting, encouraging, and challenging me since I started performing research. Thanks to Ali-Reza Adl-Tabatabai, Tim Harris, Mark Holmes, Hartmut Kaiser, Victor Luchangco, Mark Moir, Gilles Pokam, Graham Price, Randy Richmond, Paul Rogers, Michael Scott, Nir Shavit, Tatiana Shpeisman, J Smart, Win Smith, and Shane Zumpf. I would like to send a special thanks to Michael Spear, who was there to help me learn about transactional memory before my first publication and before I had built my first software transactional memory system. Thank you Michael, your early mentorship helped lay some important foundations in my understanding of transactional memory. I also would like to send out a special thanks to Dwight Winkler for his numerous points of feedback; you have been a tremendous editor Dwight and this work has been notably improved because of your precision.

Finally, and most importantly, I would like to thank my wife, Lori. Lori, without you I never would have returned to academia. You encouraged me and opened a new world of possibilities to me. Words cannot express how grateful and indebted I am to you for what you have done for me. I cannot imagine life without you. Thank you, Lori, you are my inspiration.

Contents

Chapter

Figures

Figure

Chapter 1

Introduction

Software programs can be divided into two broad categories with regard to instruction execution. In general, these two categories consist of *sequential programs* and *parallel programs*. Sequential programs generally execute one instruction at a time, while parallel programs can execute multiple instructions simultaneously. [1] Sequential programs tend to be easier for programmers to reason about than their equivalent parallel programs [51], yet, sequential programs cannot generally utilize the resources of more than one computational core. Because of this and because of the trend away from single-core machinery, it is unlikely that sequential programs will yield linear performance improvements with next generation processors as they have in the past.

To address this concern, programmers are writing parallel programs to utilize more of the resources that are available in current and next generation chip multiprocessors (CMPs) [64]. A major obstacle prohibiting the correct and efficient development of parallel programs is that the use of traditional synchronization mechanisms, such as locks, monitors, and barriers, are error-prone and are often misunderstood by even the most competent programmers [51, 77]. Due to this, researchers in the field of parallel computing have turned their attention to *transactional memory* (TM), a concurrency control paradigm that hides some of the concurrency control complexities that are exposed in traditional synchronization mechanisms. Researchers have found that TM is less error-prone than locks and can notably simplify writing correct parallel software [77].

While TM is an appealing alternative to existing synchronization mechanisms because it

[1] In fact, instruction level parallelism exploited by out-of-order pipelined or in-order VLIW superscalar microarchitectures can execute multiple instructions of a sequential program simultaneously [44].

simplifies writing parallel programs, there is uncertainty in the research community with regard to TM's performance. Some show that TM is quickly approaching the efficiency of locks [15], while others question whether TM is a viable alternative to fine-grained locking and usable in any practical development environment [12]. In this work, we show that TM performance can be improved beyond the current state-of-the-art for certain workloads by using a technique called invalidation. Furthermore, we show that invalidation can provide enforceable TM execution guarantees, such as fairness, and improve the state-of-the-art performance for programs that combine pessimistic and optimistic critical sections. We also present a new theoretical framework in which we prove that invalidation fulfills certain correctness requirements.

The remainder of this chapter provides a short overview of the different types of parallel programs and why such programs are generally written. We briefly discuss the shortcomings of the current synchronization mechanisms that are used to write parallel programs today and demonstrate how TM avoids some of these shortcomings. We then present an overview of some of the core areas of TM which are central to the rest of this work. We conclude this chapter by listing our contributions and a road map for the remaining chapters.

1.1 Parallel Programs for Convenience and Performance

Parallel programs are generally written for convenience or performance [21, 55, 70]. Parallel programs are sometimes written out of convenience because problems that are naturally stand-alone can be pulled apart from the main program and written as a separate unit. By separating these problems away from the main application, they tend to become easier to reason about and can result in reduced logical complexity [21, 55, 70]. Examples of such convenience problems are those that naturally block, such as network socket processing and user-interface interaction, which if solved inside the main body of a program can be more challenging to program due to the introduction of polling loops.

The other type of parallel programs, those created for performance, can be developed using various methods. One way to develop performance-driven parallel programs is to use multiple

threads to execute instructions simultaneously. These parallel programs, called *multithreaded* programs, exploit task parallelism [54], also known as thread-level parallelism (TLP) [44]. Unlike instruction-level parallelism (ILP), where compilers and processors automatically identify and execute parallel instructions, task parallelism relies on programmers to manually specify workloads that can be run concurrently. [2]

Multithreaded programs achieve task parallelism by using threads to execute portions of the program's workload simultaneously. At specific points during the program's parallel execution, the threads communicate the results of their work through data that are visible to all threads of the program. Such data are referred to as *shared data* or *concurrent objects* because all threads within the multithreaded program have read and write access to these objects [45, 51]. Unfortunately, a disadvantage of parallel programs is that the correct synchronization of concurrent objects is often more complex than programmers realize due to system behaviors outside of the programmers' field of view such as optimizing compiler instruction reorderings and cache coherence protocols.

1.2 Synchronization Mechanisms

For multithreaded programs to produce results that are meaningful, shared data must be accessed in a manner that is correct. We can intuit that if no data is shared between threads in a multithreaded program, the behavior will be the same as if all threads were executed one after another in a sequential version of the same program. This behavior seems intuitively correct; we can reason about how the parallel program behavior and final state will match that of the sequential program. But what if data is shared between the threads of a parallel program? How can we define correctness in terms of accessing these shared data? While we will define correctness precisely in later chapters, for the time being we can think of parallel program correctness in terms of a sequential execution of the same program.

For example, if we assume the sequential execution of a program, also called a *serial* execution,

[2] Speculative Multithreading, also known as Thread-Level Speculation, achieves TLP without requiring the programmer to specify parallel workloads [64].

is correct then a multithreaded version of the same program that produces results that are equivalent to some sequential execution of the program is also correct. In fact, this general method, called *serializability*, is a fundamental way in which correctness is measured for parallel and distributed programs [68].

Thread *synchronization mechanisms* are structures that are used to serialize access to concurrent objects in order to ensure data consistency [13, 50, 51]; in other words, synchronization mechanisms ensure parallel programs are serializable. An ideal synchronization mechanism might be one that is (1) easy for programmers to use and (2) performs competitively when compared to other synchronization mechanisms. [3]

The most common type of thread synchronization is mutual exclusion. *Mutual exclusion* is a concurrency control paradigm that creates a guarded region of program operations whose access is controlled by a variable called a lock [20] (or sometimes a monitor [52]). A *mutual exclusion lock*, or just lock, is a synchronization mechanism that is used to implement mutual exclusion. A lock is acquired and released by a thread. Once a thread has acquired a lock, no other threads can acquire the same lock until it is released by the thread that has already acquired it. The mutual exclusion control structure guarantees that a region of code guarded by a lock, or *pessimistic critical section*, is limited to a single thread of execution. Mutual exclusion provides serializability by limiting access to shared data to a single thread at a time.

Mutual exclusion locking is divided into two fundamental categories: (1) fine-grained locking and (2) coarse-grained locking [51]. When fine-grained locking is used, programmers try to enclose the smallest unit of operations performed on shared data within a critical section that is controlled by a single lock. By doing this, locks are only acquired when data they wrap is accessed, ultimately improving concurrent performance. Fine-grained locking performs well when compared to other synchronization mechanisms, but is notoriously difficult to develop, verify, and maintain. In contrast, coarse-grained locking works by using a single lock to guard multiple code regions.

[3] This is a broad generalization, but is meant to demonstrate that intuitive use and competitive performance are generally important criteria for all synchronization mechanisms.

Coarse-grained locking is easier to implement and verify than fine-grained locking, yet it can perform notably slower than its fine-grained locking counterpart. Mutual exclusion therefore seems less than ideal because it can deliver performance or simplicity, but generally not both.

To demonstrate the complexity created by fine-grained locking, consider the multi-threaded program execution shown in Figure 1.1. When both threads execute serially (one after another), the program result and behavior is correct with regard to what a programmer would expect if the program was executed sequentially. However, when the execution is interleaved as shown in Figure 1.1, the program will *deadlock* a condition where each of the program's threads are waiting for resources that are controlled by others threads preventing the program from making forward progress [51].

```
1   //---------------------------
2   // Thread 1
3   //---------------------------
4   lock(X); // acquire lock X
5   ++x;
6
7                                       //---------------------------
8                                       // Thread 2
9                                       //---------------------------
10                                      lock(Y); // acquire lock Y
11                                      lock(X); // acquire lock X
12                                      int tmpY = y;
13                                      int tmpX = x;
14                                      unlock(X);
15                                      unlock(Y);
16
17  lock(Y); // acquire lock Y
18  ++y;
19  unlock(X);
20  unlock(Y);
21
```

Figure 1.1: Two threads using fine-grained locking.

1.3 Transactional Memory

Transactional memory (TM) is a modern concurrency control mechanism that uses *transactions* as its synchronization mechanism [50, 54]. A transaction is a finite sequence of operations

that are executed with some degree of atomicity, isolation and consistency (ACI) [50, 51, 54]. The degree of ACI is based on the specific TM, which will be explained in more detail later in this chapter. TM, as proposed by Herlihy and Moss, offers several advantages over other parallel programming abstractions [50]. Two of these advantages are TM's open-ended execution model and its reduction of parallel programming complexity.

Transactions have been shown to be easier for programmers to reason about than other synchronization mechanisms because TMs move the complexity of shared memory management into the underlying TM subsystem, removing such complexity from the programmer's view [77]. Moreover, TM exposes a simplified programmer interface, reducing (or in some cases, eliminating) the potential for deadlock, livelock, and priority inversion. With some other synchronization mechanisms, such as locks, monitors, and barriers, some of the parallel programming hazards listed above may be unavoidable (as demonstrated in Figure 1.1).

Programmers use TM to create transactions that wrap regions of code that access shared data that might conflict with another thread. The operations that make up a transaction are called *transactional operations*. Transactional operations are generally rolled back when a transaction is aborted and are invisible to other transactions until the specific transaction that contains them commits. Those operations that are inside of a transaction, but are not automatically rolled back when a transaction is aborted, or have visibility side-effects, such as being immediately visible to other transactions, are called *non-transactional operations*. Transactions are commonly represented in software by the `atomic` structure shown in Figure 1.2, which replicates the prior example shown in Figure 1.1, yet the transaction example does not suffer from the deadlock condition that existed in the locking example [39].

Once a transaction has started it either commits or aborts [82]. The operations of committed transactions are seen by other threads under the illusion of occurring at a single instance in time. Therefore, the instructions within a committed transaction are viewed as if they occurred as an indivisible event, not as a sequence of operations executed serially. The operations of an aborted transaction are never seen by other threads, even if such operations were executed within a trans-

```
1   //--------------------------
2   // Thread 1
3   //--------------------------
4   atomic // Transaction T1
5   {
6      ++x;
7                                        //--------------------------
8                                        // Thread 2
9                                        //--------------------------
10                                       atomic // Transaction T2
11                                       {
12                                          int tmpY = y;
13                                          int tmpX = x;
14                                       }
15
16     ++y;
17  }
```

Figure 1.2: Two threads using transactions.

action [6]. When a transaction reads memory, such as performed in transaction T_2 in Figure 1.2, the memory elements are added to the transaction's *read set*. Likewise, when a transaction writes to memory, such as performed in transaction T_1 in Figure 1.2, the memory elements are added to the transaction's *write set*.

Consider the case of Figure 1.2, which is identical to Figure 1.1, except that the example uses transactions instead of locks. When transaction T_1 commits, both operations `++x` and `++y` are made visible to other thread at the same instance in time. If transaction T_1 aborts, neither operation, `++x` nor `++y`, would appear to have been executed even if the local transaction executed one or both operations. Furthermore, if transaction T_2 commits before transaction T_1, T_2 would see the values of `x` and `y` before T_1 executed, even though `++x` may have been executed in T_1 before T_2 committed. If the opposite were true, where transaction T_2 committed after transaction T_1, transaction T_2 would see the values of `x` and `y` after T_1 executed. In either case, the resulting values that transaction T_2 sees are consistent with some serial execution: either T_1 happens before T_2 or T_2 happens before T_1, no interleaved execution of T_1 and T_2 is returned, even though the operation `++x` within transaction T_1 happens before T_2 begins. Part of the benefit of TM is that shared data complexity, such as the complexity found in the above example, is automatically managed by the

TM subsystem allowing the programmer to focus on other issues.

In addition to reducing the complexity of parallel programming, TM has an open-ended execution model which allows it to be implemented so transactions are executed speculatively (also known as optimistic critical sections). An advantage of speculative transaction execution is that if no conflicts exist between transactions, the transaction throughput can be increased over what might be possible with a pessimistic transaction implementation. In turn, this can reduce a program's total execution time. With other synchronization mechanisms, such as locks and monitors, speculative execution might not be permitted. In a TM when concurrently, and speculatively, executing transactions do not exhibit shared data conflicts these transactions can be managed such that they are guaranteed to commit.

1.3.1 Types of Transactional Memory

At the highest level, there are three types of TM: software, hardware, and hybrid. Software transactional memory (STM) uses software to implement TM, hardware transactional memory (HTM) uses hardware, and hybrid transactional memory (HyTM) uses both software and hardware [54]. To date there has been less focus on HyTM than STM or HTM. STM and HTM have fundamentally different advantages and disadvantages.

HTM is generally believed to be faster than STM due to the prototypical designs using static random access memories (SRAM) for transaction implementations that yield roughly two orders of magnitude improved response-time over the dynamic random access memories used (DRAM) in STM designs [44]. To date HTMs have been theoretical (e.g., based on simulation) with two exceptions: the Rock Processor by Sun Microsystems and Vega 2 by Azul Systems, both of which are fabricated processors with hardware-level TM support [16]. While HTMs are believed to outperform STMs, HTMs generally require some degree of STM support due to the limitations in the number of hardware memory elements available for transactional memory accesses. To avoid this dependency on STM, some HTMs propose radical chip redesigns in order to support transactions of unlimited size without spilling to software [6].

STM is bounded only by the machine's virtual memory space and therefore can theoretically run transactions of near unlimited size. However, STMs generally come with a degradation of performance when compared to HTM. Certain STMs also require added syntactic source code changes to permit transactions that are generally not required by HTMs [26, 58]. However, STMs are notably easier to prototype and verify than HTMs [54]. Because of these reasons, STMs have received wider attention than HTMs to date [40].

1.3.2 Transactional Properties

Within a transactional memory system, transactions are provided with some degree of atomicity, consistency, and isolation (informally defined below). While the atomic, isolated, and consistent (ACI) characteristics of TM transactions are derived from database (DB) transactions, their meaning in a TM system may differ from that of a DB system. A key difference in TM ACI from DB ACI is that consistency in a DB system is generally defined in terms of maintaining the integrity constraints of the DB, such as ensuring invariants are not violated, like the preservation of foreign key constraints [34, 38]. Consistency in a TM system generally encompasses the notion of serializability, where a TM execution can be loosely defined as serializable if its total commit order and conflicting transactional operations maps to some serial execution of the same committed transactions and conflicting transactional operations in some precise way. TM consistency is also sometimes extended to include the preservation of integrity constraints, such as those captured by the property of opacity defined by Guerraoiu et al. (to be discussed in more detail later in this chapter) [36]. Informally, ACI in a TM context has the following general meaning:

- Atomicity – The operations of a transaction appear to happen instantaneously. When a transaction commits, all of its operations are made visible to the external world at a single instance in time.

- Consistency – Transactional executions must meet some minimal notion of serializability, such as final-state, view, or conflict serializability [68]. Loosely speaking, the transactional

execution of committed transactions and transactional operations should map, in some precise and meaningful way, to a serial execution of the same committed transactions and transactional operations. Final-state, view, and conflict serializability all define slightly different semantics with regard to how transactional executions map to serial executions.

- Isolation – Transactions must execute in isolation; when a transaction is *active* – the transaction has started but has not yet committed or aborted – its transactional operations must not be visible to any other transaction or external viewer. [4] In other words, the transactional operations of any active transaction must not be visible to any external observer.

1.3.3 Conflict Detection

Conflict detection is the process of determining if a transaction can commit [81, 84]. There are many ways to perform conflict detection. Two of the primary ways are through *validation* and *invalidation*. In a broad sense, a TM that performs validation can be thought of in the following way. First, each piece of transactional data has an associated version number with it. When a transaction reads or writes data, it stores an associated version number with that data. When a transaction commits, its written data is updated in the globally shared area that all transactions can access, and the version numbers associated with that data are incremented. When a transaction performs conflict detection to determine if it can commit, the TM verifies that the version numbers of the committing transaction's read and write data are the same as the version numbers of the same memory that stored in the program's globally shared space. If a version mismatch is found, it means another transaction has already committed and wrote to memory that the committing transaction is accessing. When a version mismatch is found, the committing transaction must abort.

A TM that performs invalidation uses a fundamentally different principle for conflict detection and can be thought of in the following manner. First, transactional memory in an invalidating TM does not use version numbers. Instead, each transaction has a *valid* flag which is initialized to true

[4] Active transactions are also known as in-flight transactions.

when the transaction becomes active. When a transaction, T_a, begins its conflict detection phase, the TM looks for overlaps between T_a's write set and all other active transactions, $T_b ... T_z$, read and write sets. If an overlap is found, the TM generally forces either T_a to abort or the other active transactions that have an overlap with T_a in their read or write set. In the following chapters of this work, we provide a detailed analysis of the benefits and drawbacks associated with a TM that performs invalidation.

At a high level, an important distinction between validation and invalidation is that a TM that performs validation does not need to analyze active transactions to commit. On the other hand, a TM that performs invalidation does not need to analyze globally shared memory to commit. These distinctions can result in notable performance differences in *transaction throughput*, the number of transactions that commit per second, which are more fully described in the subsequent chapters.

1.3.3.1 Eager and Lazy Conflict Detection

Conflict detection can be performed at various points throughout a transaction's lifetime [54, 84]. In general there are two distinct points at which conflict detection can be performed: at *conflict-time* (also known as eager conflict detection), the point in time where two transactions access the same piece of memory, or at *commit-time* (also known as lazy conflict detection), the point at which a transaction begins its commit phase. Most TMs require conflict detection be performed lazily, even if the TM also performs conflict detection eagerly. The prior research of Marathe et al. has shown that performing conflict detection with *non-blocking TMs* at various times, in addition to commit-time, can yield performance benefits [58]. [5] However, in *lock-based TMs*, Dice et al. found that commit-time only conflict detection can perform better than other alternatives [17]. [6]

[5] Non-blocking TMs are TMs that use non-blocking atomic primitives, such as LL-SC or compare-and-swap, to provide TM ACI.

[6] Lock-based TMs are TMs that use mutual exclusion locks to provide TM ACI.

1.3.4 Opacity

While TMs perform conflict detection to determine which transactions can commit, as noted by Guerraoui and Kapalka, conflict detection alone may not be sufficient for TMs to ensure program correctness. Guerraoui and Kapalka note that TMs may also be required to perform *opacity*, a correctness criterion that requires each transactional read return a value that is consistent with its execution, and that doomed transactions be aborted before a subsequent transactional read or write returns from its call [37]. Guerraoui and Kapalka show that even an isolated, uncommitted transaction can cause adverse program side-effects if a single transactional read is inconsistent.

```
1   //--------------------------
2   // Thread 1
3   //--------------------------
4   atomic // Transaction T1
5   {
6      if (y == 0) return 0;
7                                        //--------------------------
8                                        // Thread 2
9                                        //--------------------------
10                                       atomic // Transaction T2
11                                       {
12                                          x = 0;
13                                          y = 0;
14                                       }
15
16     return x / y;
17  }
```

Figure 1.3: A Simple Example of Opacity.

To demonstrate this, consider the program shown in Figure 1.3, which we will suppose has a *program invariant* (some fact about the data of the program), where variables `x` and `y` are always both zero or non-zero. As shown in Figure 1.3, transaction T_1 checks the value of `y` to ensure it is not zero. If it is zero, T_1 immediately returns. This is done to ensure a divide by zero operation is not executed in the following line. However, if transaction T_2 is allowed to execute from beginning to end after transaction T_1 has checked the value of `y`, the operation `x / y` will emit a divide by zero exception, even if T_1 would have eventually been aborted due to its conflicting access of

variables `x` and `y` with transaction T_2.

To avoid this problem, a TM performs opacity checks each time a new memory element is read. In this case, when transaction T_1 reads `x`, it performs an opacity check to ensure all previously read data elements are still consistent with their original state. For T_1, the opacity check on `y` would immediately result in a failed opacity check causing T_1 to abort before it could perform the `x / y` divide by zero operation. Thus to be correct, in addition to detecting and resolving conflicts, TMs may also need to be opaque.

1.3.5 Direct and Deferred Update

Updating is the process of committing transactional writes to globally shared memory and is performed in either a direct or deferred manner [54]. *Deferred update*, also known as lazy update or lazy write acquisition, creates a local copy of shared data and performs the transactional write operations to the local copy. If the transaction commits, the TM sends the local modifications to the globally shared memory space. If the transaction aborts, no additional work is done because all writes were performed on temporary data. Deferred update is sometimes said to use a *redo log*, because the transactional write operations are redone if the transaction commits.

Direct update, also known as eager update or update-in-place, constructs a backup copy of the globally shared data and then writes directly to the shared data. If the transaction commits, the transaction performs no additional work to commit the written data. This is because the written data is already placed in the globally shared space. If the transaction aborts, the transaction restores the globally shared data to its original form using the transaction's backup copy. Direct update is sometimes said to use an *undo log*, because when a transaction is aborted, the backup copies of the transaction's written data are used to undo the transactional write operations.

Some TM systems use direct update because, among other benefits, it optimizes the common case of transactional commits and does not require a layer of indirection to perform writes (BSTM [42], McRT-STM [78] and LogTM [60]). Other TM systems favor deferred update because it does not prematurely limit concurrent writer and reader transactions on the same data and be-

cause it supplies contention managers (CMs) with a more complete view of transactional data sets, among other reasons (InvalSTM, TL2, and RingSTM) [33, 17, 19, 83].

1.3.6 Ownership and Non-Ownership Records

One way in which memory is associated with a transaction that accesses it, is through the use *ownership records*. In general, ownership records work by constructing a transactional list to each piece of shared memory that can be accessed by a transaction [82]. When the shared memory is accessed, some identifier for the transaction is stored in the memory's associated transactional list. When the transaction commits or aborts, the transaction is removed from the memory's associated transactional list. At any instance in time, many transactions may be on a single memory's transactional list.

A shortcoming of ownership records is that they usually create some serialization point on the memory being accessed by a transaction in order to add or remove transactions to the transactional lists associated with the memory. As such, only one transaction can usually modify the transactional list for a given memory at a time. Because of this, researchers have explored *non-ownership records*, where memory is associated with transactions without the need to annotate memory directly and, likewise, without the need to serialize access to such memory [14, 33]. However, when ownership records are not used, other system limitations can arise such as transactions being unable to see other transaction's read or write sets which can limit or even prevent certain types of behaviors (e.g., invalidation-based conflict detection).

1.3.7 Strong and Weak Isolation

Isolation, as part of the ACI properties above, can be implemented strongly or with various degrees of weakness. The difference in implementation results in substantially different behaving TM systems. A TM that implements strong isolation requires that each non-transactional operation behave as an individual transaction. Strongly isolated systems disallow non-transactional operations from seeing into a transaction's execution. The only visible states of transactional op-

erations are before they begin and after they commit. Due to this, it is theoretically impossible to have inconsistent shared data behaviors in strongly isolated systems. However, strong isolation can have enormous performance overheads and are generally very challenging to implement entirely in software. As such, strong isolation tends to require some hardware support which may (or may not) be an unrealistic requirement.

A TM that implements weak isolation guarantees that transactional operations within an active transaction are not visible to other active transactions. However, non-transactional operations within active transactions are visible to other transactions or other threads, as are operations that are entirely outside of transactions. Due to this, weakly isolated TM systems are susceptible to shared data inconsistencies. TM designers of weakly isolated systems place the responsibility of program correctness upon the programmer. In general, as long as the programmer follows some set of rules with regard to transactional and non-transactional operations, as well as operations entirely outside of transactions, the TM will behave consistently. While weakly isolated systems are prone to program correctness problems, they are also easier to implement and can perform notably faster than their strongly isolated counterparts.

1.4 Contributions

The following lists the primary contributions of this work presented in the following chapters and the methods used for evaluating each contribution.

1.4.1 Contribution I (LCSD'07 [26], CGO'10 [32])

We empirically show that, for programs with notable contention or for programs that execute transactions with many memory elements, an STM that uses full invalidation can result in greater transaction throughput than an STM that only uses validation or uses a hybrid of validation and invalidation.

Approach. To fulfill this contribution, we have analyzed several benchmarks. In particular, we have analyzed programs with and without significant contention and programs with few and many memory elements. In both cases we have compared our STMs, which only use invalidation (InvalSTM and DracoSTM), against STMs that only use validation (TL2) or use hybrid validation and invalidation for each transaction (RSTM).

Results. Our initial results compared DracoSTM, our first invalidation-only STM, to RSTM, an STM that uses a hybrid of validation and invalidation conflict detection for each transaction. Our results suggest that for workloads with few memory elements, a hybrid validation-invalidation STM can outperform an invalidation-only STM. For workloads with many memory elements, an invalidation-only STM can outperform a hybrid STM that uses both validation and invalidation [26].

More recently, we compared InvalSTM, our most recent and efficient invalidation-only STM, to TL2, a validation-only STM and that is widely considered a state-of-the-art STM with regard to performance [32]. The results in these experiments suggest that an invalidation-only STM can outperform a validation-only STM if there is contention in the program that can be managed in a way that increases transactional throughput. Our results also show that in transactions with few memory elements and little contention between those memory elements, a validation-only STM can outperform an invalidation-only STM.

1.4.2 Contribution II (PODC'08 [28], CGO'10 [32])

We demonstrate through complexity analysis that (1) invalidation has smaller algorithmic growth rates than validation as transactions access more memory and that (2) the contention manager in an invalidating TM can increase transaction throughput over a contention manager in a validating TM by making informed decisions that reduce the number of transactional aborts or allows longer running transactions to commit thereby reducing the execution time of the critical path.

Approach. To fulfill this contribution, we provide a mathematical analysis of the overhead incurred by state-of-the-art STMs that use validation conflict detection against our fully invalidating STM. We use benchmarks to empirically demonstrate the efficiency of invalidation's conflict detection for transactions that access many memory elements. For contention managers, we analyze the information provided to the contention manager and demonstrate that the contention managers of validating TMs receive less information than those of invalidating TMs. We then show that this reduced information can result in less efficient decision making with validating TMs, which results in reduced transaction throughput and longer execution times.

Results. We have explored the asymptotic and experimental analysis of invalidation compared to validation [32, 28]. Our initial experimental analysis substantiates our theoretical and analytical claims that invalidation is more efficient than validation for memory-intensive transactions or executions that contain notable contention. As transactional contention increases or the memory that transactions access increases, the performance divide between invalidation and validation continues to grow at a superlinear rate and invalidation becomes more efficient compared to validation. However, for transactions that access few memory elements and contain little contention between them, both our analytical and experimental results show that a validation-only TM may have superior performance when compared to an invalidation-only TM.

1.4.3 Contribution III (EPHAM'08 [27])

We demonstrate that invalidation has greater transaction throughput than validation when transactions with user-defined priority must be preserved.

Approach. To fulfill this contribution, we have analyzed the system requirements where user-defined priority-based transactions exist and how user-defined priority-based transactions execute alongside non-prioritized transactions. We show that invalidation does not reduce transaction throughput, while validation does by falsely identifying conflicts in order to ensure transactional priority is preserved.

Results. Our results demonstrate that invalidation is more than 100× more efficient than validation for TMs that support user-defined priority-based transactions [27]. We have demonstrated that validating TMs can support user-defined priority-based systems but at the cost of many false positives on transactional conflicts making their performance suffer compared to invalidation.

1.4.4 Contribution IV (ASPLOS'09 [25], ICOOOLPS'09 [31])

To show that invalidation resolves some lock-aware TM (LATM) problems – such as transitioning from optimistic transaction-based critical sections to pessimistic lock-based critical sections and performing unified contention management – without requiring additional complexity and overhead as needed in the most efficient validating LATM implementations.

Approach. We have analyzed basic characteristics of LATM and demonstrated through algorithmic analysis that invalidation can handle certain LATM problems with less complexity and more efficiently than a validation-only manner. We have further shown that to support irrevocable transactions, as may be necessary for LATM implementations, TMs must support some form of invalidation. We have also briefly explored what we call *unified contention management*, a contention management system that handles both optimistic and pessimistic critical sections, and have found that invalidation has some benefits over validation.

Results. We have built an efficient invalidation-only LATM that is competitive with the state-of-the-art LATMs from Volos et al. and Ziarek et al. We have demonstrated that for locks outside of transactions (LoTs), our LATM can be more efficient than Volos et al.'s TxLocks under certain circumstances. We also demonstrated that for locks inside of transactions (LiTs), our LATM can be more efficient than Ziarek et al.'s Atomic Serialization under certain circumstances [25, 31].

We have also performed research which simplifies the interaction of transactions and locks [29]. This research has focused primarily on simplifying the programmatic overhead of locks when used

in conjunction with transactions. We also have some early results in our exploration of unified contention management.

1.4.5 Contribution V (WTTM'10 [30])

We prove that full invalidation is correct in terms of conflict and view serializability. In particular, we show that if a history is accepted by a full invalidation automaton then that history is both conflict and view serializable.

Approach. To fulfill this contribution, we have modified Christos Papadimitriou's formal model on concurrency theory found in, *"The Theory of Database Concurrency Control,"* and use these modified data structures in our proofs [68]. In addition, we extend Lynch et al.'s I/O automata theory found in, *"Atomic Transactions: In Concurrent and Distributed Systems,"* and define full invalidation in terms of an I/O automaton.

Initial Results. Our initial results and research directions were published at the 2010 Workshop on the Theory of Transactional Memory (WTTM'10 [68]). In addition to proving full invalidation is conflict and view serializable, we have constructed a new theoretical framework to represent deferred update TMs. While Papadimitriou's work on conflict graphs has been fundamental to our research, we have found that these graphs unnecessarily limit concurrent throughput and that certain optimizations found in our system could not be properly represented with this graphs.

To address this limitation, we have constructed a new type of conflict graph, which we call a *lazy conflict graph*. Lazy conflict graphs relax restrictions that exist in Papadimitriou's conflict graph that unnecessarily limit concurrent transaction executions that may be legal in a deferred update system. By relaxing restrictions in these graphs to support behaviors that are legal in certain deferred update TMs, we have discovered new optimizations that exist in a full invalidation. We have integrated these new optimizations into InvalSTM and they will be available in the next version of the system.

1.5 Road Map

The road map of this work is as follows. Chapter 1 presented an overview of TM, including its advantages and disadvantages compared to other parallel programming concepts.

Chapter 2 begins by presenting conflict detection, an active area of research in TM optimizations. We then present new methods for optimizing TMs using invalidation, which in some cases are more efficient than the prior state-of-the-art systems. Chapter 3 follows by formalizing the STM implementation of Chapter 2. In Chapter 2 and prove that a history that is accepted by a full invalidation automaton is both conflict and view serializable.

Chapter 4 and 5 focus on the importance of an invalidation-based STM for both efficiency and simplicity. Chapter 4 shows that an invalidation-based STM can outperform a validation-based STM by upwards of 100× for systems that must respect user-defined priority-based transactions. Chapter 5 demonstrates that in order for transactions and locks to work together in the same program, some form of invalidation may be necessary. Furthermore, Chapter 5 demonstrates that if a TM uses invalidation, it may reduce the system's overall complexity when making that TM lock-aware.

Chapter 6 presents a brief conclusion of this work. We discuss the practical performance of an efficient implementation of full invalidation, the theory of full invalidation, priority-based transactions in a full invalidation TM, and lock-aware TM using full invalidation.

Chapter 2

Optimizations: Transactional Contention and Memory-Intensive Transactions

Many TMs use an optimistic concurrency model in which transactional operations are executed concurrently and the operations that violate the TM's supported serializability [50, 67] are undone. Some degree of computational overhead is incurred when TMs provide an optimistic concurrency model where the transaction commit order is serializable, and the transactions themselves are atomic and isolated. Furthermore, some researchers argue that this incurred computational overhead is too great for TM to be practical [12]. To address these concerns, researchers have found innovative ways to reduce the overhead of TMs by optimizing conflict detection [17, 19, 59, 74, 86, 84].

Conflict detection, the process of determining if transactions can commit [84], is usually implemented as a conservative overestimation of some form of serializability [68]. A transaction can commit as long as its commit preserves some precise definition of serializability with regard to the total transaction commit order. Serializability is usually preserved by disallowing some number of transactional conflicts from existing between committed transactions. A *transactional conflict* is loosely defined as the non-null intersection between one active transaction's write elements and another active transaction's read and write elements [50, 60, 74]. While significant work has been done in the area of conflict detection and resolution, nearly all TMs perform *commit-time validation*, a strategy where a single transaction's read elements, and sometimes its write elements, are checked for consistency during a transaction's commit phase.

Commit-time validation typically uses version numbers associated with memory to track transactional conflicts [78]. In general, the version numbers of a transaction's read and write

elements (also known as *read* and *write sets*) are compared against the version numbers of the same memory stored, and shared, globally. If a version mismatch is found, the validating transaction is aborted because a previously committed transaction has updated the same memory. If no mismatch is found the transaction is consistent and can be committed. While commit-time validation is efficient for workloads that exhibit little contention, it limits *transaction throughput*, the number of transactions that commit per second, for contending workloads. This is because commit-time validation does not determine how many in-flight transactions will be aborted due to a transaction's commit.

In this chapter we analyze the differences between commit-time validation and commit-time invalidation for transactions that access many memory elements and for workloads where notable contention exists between transactions. *Commit-time invalidation* is an implementation of the full invalidation, where full invalidation is a specification for conflict detection which is defined precisely in Chapter 3. In general, commit-time invalidation finds transactional conflicts by comparing the memory of a committing transaction against the memory of in-flight transactions. Commit-time invalidation differs from commit-time validation in that all of a committing transaction's conflicts with in-flight transactions are found and resolved *before* the transaction commits. Conflicts are sent to the *contention manager* (CM), the process that decides which transactions make forward progress [36, 49, 79], for resolution. The CM resolves conflicts by either (1) aborting all conflicting in-flight transactions, (2) aborting the committing transaction, or (3) stalling the committing transaction until the conflicting in-flight transactions have committed or aborted [84]. Through this mechanism, commit-time invalidation can notably increase transaction throughput when compared to commit-time validation for contending workloads.

Although invalidation is not a new idea [22, 26, 39, 49, 81, 84, 83], to the best of our knowledge, no prior work has implemented an efficient TM (one that is competitive with the state-of-the-art) using only invalidation. Inefficiencies found in prior attempts have steered TM research toward validation. In this chapter we demonstrate that a TM which only uses commit-time invalidation can be implemented efficiently. In doing so, this chapter iterates through the following contributions:

(1) Full commit-time invalidation can increase transaction throughput by supplying a CM with more information than is possible using commit-time validation, allowing the CM to make informed and efficient decisions.

(2) Optimized commit-time invalidation has a slower asymptotic operational growth rate than commit-time validation with regard to the memory elements accessed within a transaction. This means that as the number of memory elements within a transaction increase, commit-time validation requires proportionally more operations to perform than commit-time invalidation.

(3) Commit-time invalidation requires zero operations to identify conflicts in dynamically detected read-only transactions and ensures opacity[1] [37] for any transaction in $O(N)$ time, where N is the number of elements in the transaction's read set, an improvement over incremental validation's $O(N^2)$ time.

(4) Our efficient commit-time invalidating STM, InvalSTM, is over 3× faster than TL2 [17], a state-of-the-art validating STM, for certain contending workloads. We further show that our earlier STM, DracoSTM, outperforms RSTM for memory-intensive transactions.

2.1 Insights into Conflict Detection

In this section we present a history of invalidation and explain why prior efforts have only partially explored it, leaving many of its powerful optimizations unexplored.

One of the most computationally expensive aspects of TM is the process of detecting conflicts (i.e., discovering when the commit of two or more transactions will result in an execution order that is not serializable).

If, for a moment, we restrict ourselves to commit-time conflict detection, we can see why an invalidating TM can exploit more transaction throughput than a validating TM. Consider the scenario depicted in Figure 2.1 where one transaction writes to variable X and N transactions

[1] Opacity is the property where doomed transactions are identified before they can execute harmful operations [37].

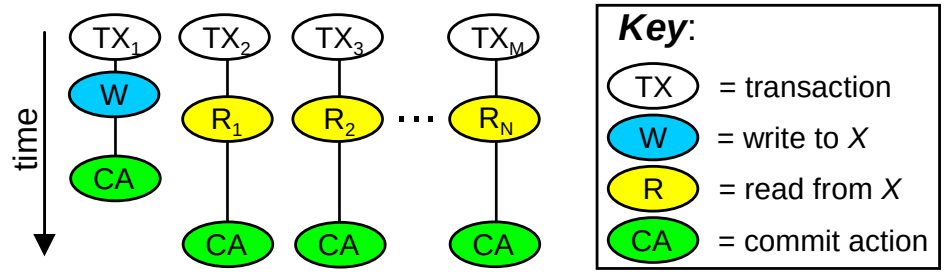


Figure 2.1: 1-Writer and N-Readers: A Highly Contending, Highly Concurrent Workload.

subsequently read the value of X. A TM using commit-time validation (see Figure 2.2 for an overview) and *lazy write acquisition*[2] , will successfully validate the writer transaction at its CA (the commit action). The writer will then update X's global value and version number and commit. However, this behavior will cause all N readers to abort. When the readers reach their CA, they will be required to abort because their view of X will be stale with regard to TX_1's commit. Thus, commit-time validation effectively eliminates all concurrency between the readers and writer, a serious issue if N is large like it can be in a number of workloads and systems [23, 65, 88].

Now consider commit-time invalidation (see Figure 2.3 for an overview) for the scenario in Figure 2.1. When TX_1 reaches CA it scans all N active transactions for conflicts. Each conflict is sent to the contention manager which, based on the number of contending *read-only transactions* (transactions that only read memory), can make an informed decision to abort TX_1 and allow the concurrent commit of the N readers. When N is large, this behavior dramatically increases transaction throughput.

Unfortunately, prior to this work, validating TMs have proven to be more efficient in practice. To understand why this is so, we must first discuss the different types of conflicts, conflict detection strategies, and the different strategies for maintaining read and write sets.

[2] Lazy write acquisition is where written memory is exclusively acquired at the transaction's commit phase [54], see Chapter 1.

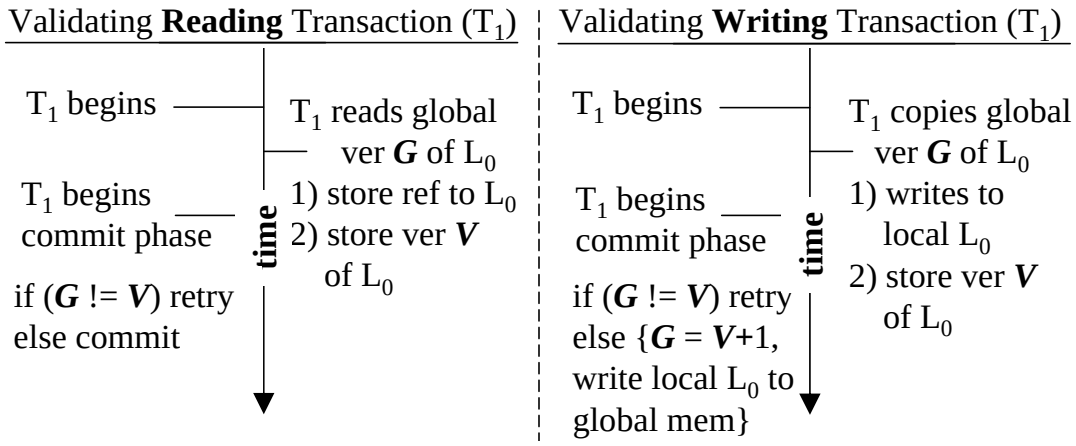


Figure 2.2: Transaction Using Commit-Time Validation.

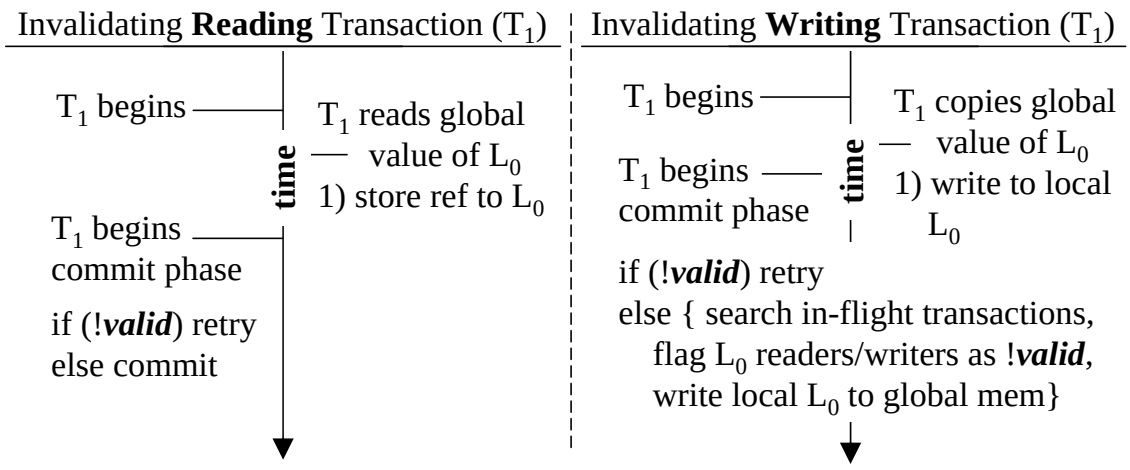


Figure 2.3: Transaction Using Commit-Time Invalidation.

2.1.1 Types of Conflicts.

Conflicts arise when two or more transactions access the same memory and come in three varieties: *W-W*, *W-R*, and *R-W* [74]. W-W conflicts occur when two transactions write to the same memory. W-R conflicts occur when one transaction writes to a memory location and another transaction subsequently reads it (or vice versa for R-W conflicts).

The way in which conflicts can be handled is specific to the TM specification and implementation. For example, dependence-aware TM [73] can often allow transactions with all types of conflicts between them to commit as long as their commit order follows the order in which the conflicting events (W-W, W-R, and R-W) were ordered in the execution.

TMs generally maintain read and write sets (i.e., the set of locations a transaction has read and written, respectively) to detect conflicts between transactions. A transaction's read or write set is said to be *visible* if it can be seen by other transactions, otherwise it is called *invisible*. For invalidating TMs to detect W-W conflicts between active transactions, write sets must be visible so that write sets from different transactions may be compared. Likewise, for invalidating TMs to detect W-R or R-W conflicts, read sets must be visible. For validating TMs to detect W-W, W-R, or R-W conflicts, read and write sets do not need to be visible. This is because validating TMs identify conflicts by comparing the versions numbers of their read and writes against the version numbers of the same globally, shared data.

As noted by prior research, enabling visibility for read and write sets can incur substantial overhead [84, 87]. Visibility for read and write sets has since been seen as a feature in which the cost of the additional computations needed to enable visible read and write sets can outweigh the benefits of read and write set visibility [84]. Later in this chapter we give a special treatment to the manner in which we optimize the necessary computations to provide the efficient visible read and write sets that are necessary for invalidation.

2.1.2 The Rise and Fall of Invalidating TMs

Partial invalidation is a process in which a TM performs some invalidation, either eagerly or lazily, but does not guarantee all conflicts with active transactions will be resolved before a transaction commits. Because some conflicts can be missed by invalidation, partial invalidation subsequently requires that some transactions perform some degree of conflict detection through validation. Partial invalidation was first implemented in Herlihy et al.'s DSTM for eager W-W conflicts, and in Harris and Fraser's WSTM for lazy W-W conflicts [39, 49]. Scott followed by proposing several invalidation techniques, which were used in Spear et al.'s *mixed invalidation* (using eager W-W and lazy W-R / R-W invalidation) in RSTM [81, 84]. Fraser and Harris's OSTM followed by implementing lazy invalidation for W-W conflicts [22].

To maximize concurrency, these TMs use *non-blocking synchronization primitives*: they avoid the use of locks for shared data structures and instead rely on wait-free, lock-free (OSTM), or obstruction-free (DSTM, RSTM) synchronization (see Appendix for more details). To maintain the visible read and write sets needed for invalidation, they use ownership records or *orecs*, which are data structures that associate memory elements with the transactions that access them [82]. On the first read or write of each memory location, the transaction is added to the orec for that location. Upon commit, the transaction is removed from all the orecs in which it was added.

In their most efficient implementation, orecs are computationally expensive. Spear et al. explain that maintaining complete visible readers per transactional memory location, necessary for W-R / R-W invalidation, can incur too much overhead for a TM to be practical [84]. They found the overhead associated with managing such readers costs more than the $O(N^2)$ overhead of *incremental validation*, the process of revalidating all of a transaction's read elements each time a new memory location is opened for reading (i.e., when a memory location is first read) [54, 84].

To gain some of the benefits of invalidation without incurring the full penalty of orecs, Spear et al.'s mixed invalidation uses one word per memory location to track readers. This reduces maintenance overhead, but limits W-R invalidation to 32 conflicts (or 64 on a 64-bit architectures)

per memory location. Thus, the system must still perform version-based validation for > 32 threads.

Some *lock-based* STMs (TL2, RingSTM, and JudoSTM), those STMs which use mutual exclusion operations at their core, avoid the overhead of invalidation by not using it at all. TL2, for example, does not use invalidation, yet through its space and time optimizations, is able to perform efficient orec-based validation [17]. RingSTM and JudoSTM, on the other hand, do not use invalidation nor do they use orecs. RingSTM uses a —ring— structure to efficiently perform eager validation only against those transactions on the —ring— in which it is necessary [86], while JudoSTM uses a *value-based conflict detection* data structure to perform validation with reduced atomic instruction overhead [63]. In all three cases, invalidation is avoided entirely.

Although invalidation was proposed as early as 2003 [39, 49], implementation overhead has kept its use limited. The remainder of this chapter is dedicated to showing how invalidation can be made efficient. Section 2.2 explores the concurrency potential of an efficient fully invalidating TM by presenting an asymptotic analysis of version-based validation in contrast with commit-time invalidation. The section shows that invalidation provides opportunities over validation that can increase transaction throughput, making it a superior decision-based conflict detection strategy. Section 2.3 then shows how we efficiently implement the data structures needed for full invalidation within InvalSTM, and Section 2.3.5 evaluates InvalSTM against TL2, a state-of-the-art validating STM.

2.2 The Promise of Full Invalidation

In this section we iterate through some of the primary benefits that full invalidation offers over validation. Specifically, we show that *any* fully invalidating TM can perform opacity and conflict detection more efficiently than a validating one. In the case of contending workloads, we illustrate how full invalidation can increase transaction throughput over validation. We also demonstrate that a fully invalidating TM that uses search time optimized read and write sets can perform conflict detection for memory-intensive transactions in notably fewer operations than what is needed for the most efficient validation techniques.

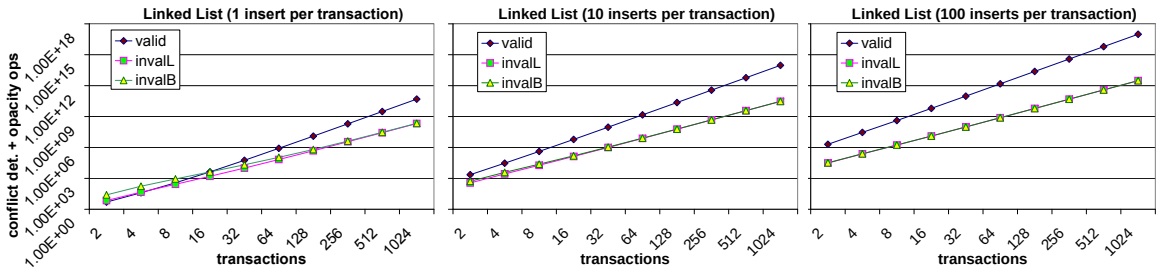


Figure 2.4: Conflict Detection and Opacity Overhead of Linked List for Validation and Invalidation.

2.2.1 Full Invalidation

For a TM to be *fully invalidating*, each conflict that a transaction has, which would make the total execution unserializable if left unresolved, must be resolved before the transaction commits. A *resolved conflict* is one that is eliminated by aborting or stalling one or more transactions to preserve serializability (as described in [67]). When a transaction begins its commit phase in a fully invalidating TM, the TM only needs to check other active transactions for conflicts against the committing transaction; a committing transaction is never checked for conflicts with previously committed transactions in a fully invalidating TM, as is necessary in a validating TM.

This is because by definition a fully invalidating TM requires that a transaction's conflicts that would make the execution unserializable be resolved before transactions commit. This behavior ensures that the only unresolved conflicts are those with uncommitted, active transactions. In other words, conflicts that would make the execution unserializable cannot exist with committed transactions. So committed transactions do not need to be checked for conflicts in a fully invalidating TM.

Although there are numerous ways to build a fully invalidating TM, commit-time invalidation seems to be necessary and performing only commit-time invalidation may be the least computationally expensive way to ensure full invalidation. As such, for the remainder of this work when we speak of full invalidation we mean a system that at least, and most often only, uses commit-time invalidation.

2.2.2 Conflict Detection and Opacity

TMs perform conflict detection to determine which transactions can commit. However, as noted by Guerraoui and Kapalka, conflict detection alone is insufficient. TMs must also ensure *opacity*, a correctness criterion that requires each transactional read return a value that is consistent with its execution, and that doomed transactions be aborted before a subsequent transactional read or write returns from its call [37]. Guerraoui and Kapalka show that even an isolated, uncommitted transaction can cause adverse program side-effects if a single transactional read is inconsistent.

To demonstrate this, consider a program invariant where variables X and Y are always one discrete value apart (+/-1). If transaction T_1 reads X as value 2, the operation X/Y will be defined, because Y will be either 3 or 1. However, if transaction T_2 performs $X = 1$, $Y = 0$ and commits after T_1 reads X but before it reads Y, the program invariant will be violated. If T_1 is permitted to read Y, the X/Y operation will result in a divide by zero exception. An opaque TM avoids this by verifying all of a transaction's reads are consistent before opening a new item for reading or writing. In this case, when T_1 opens Y for reading, the TM would identify T_1's view of X is inconsistent and force it to abort before returning the value of Y. Thus to be correct, in addition to detecting and resolving conflicts, TMs must also be opaque.

2.2.2.1 Validation

Many TMs ensure opacity and perform conflict detection using the same technique; they use *incremental validation* (DSTM, RSTM, SXM, and TL2), a process in which each element in a transaction's read set is checked for consistency each time a new element is opened for reading [84, 40]. Incremental validation performs $O(N)$ operations N times, where N is the number of elements in the transaction's read set, resulting in $O(N^2)$ operations [40, 54]. Each validation a transaction performs before commit-time is an opacity check (and still results in $O(N^2)$ worst-case time because $N * (N - 1) \in O(N^2)$). These eager operations are not intended to commit the transaction. Instead they ensure opacity by avoiding abnormal program behavior that would ensue

from accessing stale and inconsistent values. The final validation operation, performed precisely once during a transaction's commit phase, is a conflict detection operation performed specifically to ensure a transaction can commit.

Given a series of M non-conflicting, committing transactions, the below equation represents the opacity and conflict detection operations sufficient for incremental validation. The variable r_i is the ith committing transaction's read set size. [3]

$$o_v(M) = \sum_{i=1}^{M} \sum_{j=1}^{r_i} j$$

The inner sum represents the number of opacity operations performed for each transaction up to and including its commit-time conflict detection operation. The outer sum includes the opacity and conflict detection operations for all M committing transactions.

2.2.2.2 Invalidation

Invalidation uses two different techniques for opacity and conflict detection. Invalidating TMs perform opacity by checking a boolean *valid* flag, and perform conflict detection by identifying conflicts between a committing transaction and all active transactions. The invalidation processes for opacity and conflict detection are explained below.

Each transaction has a *valid* flag that is initially true. It is set to false when the TM decides to abort the transaction to resolve a conflict (Figure 2.3). When a transaction opens a new element for reading, the TM checks the transaction's *valid* flag – an $O(1)$ time operation. If it is false, the transaction is aborted before the new element's memory is returned. If the *valid* flag is true, the transaction continues to execute normally. Because opacity is checked incrementally, it takes $O(N)$ time to complete per transaction, an improvement over incremental validation's $O(N^2)$ time. However, to properly set each transaction's *valid* flag, the TM performs commit-time invalidation for each transaction. The commit-time invalidation algorithm behaves differently if the transaction is a read-only transaction or not.

[3] A TM using a global clock does not need to perform incremental validation if the global clock has the same value as when the transaction first began. This indicates no transaction has committed since it began [17]. However, the clock must still be read at each opacity check.

In an invalidating TM a *writer* transaction, a transaction that writes to at least one memory location and reads any number of locations, must resolve its conflicts before it commits. These conflicts are limited to active transactions that access memory (via read or write) that the writer has modified. When the conflicts are found the CM can perform any one of the following actions: (1) set the committing transaction's *valid* = *false* and abort it, (2) set the conflicting, active transactions' *valid* = *false* so they will abort or (3) stall the committing transaction until the conflicting transactions commit or abort. A key characteristic when using commit-time invalidation is that a committing transaction's conflicts are identified (and resolved) prior to committing; this characteristic is paramount to unlocking concurrency and will be explained in more detail in the following sections.

Given a series of M non-conflicting, committing transactions, the below equation represents the opacity and conflict detection operations sufficient for full invalidation. The variables r_i and w_i are the ith committing transaction's read and write set size. F_i is the number of in-flight transactions at the time of the ith committing transaction. r_j and w_j are the jth in-flight transaction's read and write set sizes. s_{rj} and s_{wj} are the search time complexity associated with the jth transaction's read and write algorithms.

$$o_i(M) = \sum_{i=1}^{M} \left(r_i + \sum_{j=1}^{F_i} w_i(s_{rj}(r_j) + s_{wj}(w_j)) \right)$$

The inner sum performs conflict detection for the ith committing transaction against all in-flight transactions (F_i). The number of operations sufficient to identify conflicts with each transaction is based on the jth transaction's read and write set algorithm's worst-case search time when holding r_j and w_j number of elements, represented by $s_{rj}(r_j)$ and $s_{wj}(w_j)$. The ith transaction compares its write set for overlaps with reads and writes of in-flight transactions, resulting in $s(r_j) + s(w_j)$. Finally, w_i is multiplied by the sum of $s(r_j)$ and $s(w_j)$, because each search operation is performed w_i times, the number of elements in the committing transaction's write set. [4]

[4] This is not true for algorithms that perform set intersection in constant time, such as Bloom filters. In such cases, w_i becomes some constant C.

The outer sum performs the incremental opacity checks for all M committing transactions. The checks are performed with a single boolean comparison and are performed each time the ith transaction opens a new element for reading, represented by r_i.

Notice that read-only transactions in a full invalidation TM have $w_i = 0$, which reduces the above equation to $o_i(M) = \sum_{i=1}^{M} r_i$. In other words, read-only transactions perform zero conflict detection operations. Furthermore, unlike other optimizations, such as TL2's read-only optimization, invalidating read-only transactions do not need to be identified as read-only before they execute to achieve this benefit. [5] Therefore, dynamically detected read-only transactions in any fully invalidating TM are not required to perform any conflict detection operations. This is a notable benefit because many transactions that are statically read-write may be dynamically read-only much of the time.

2.2.3 An Analysis of Opacity and Conflict Detection Efficiency

To demonstrate the efficiency of opacity and conflict detection using full invalidation, consider a scenario in which N transactions are appending to a linked list in which only one transaction can commit. For this scenario, each time a transaction commits we assume that all other active transactions must abort and restart, regardless of the conflict detection strategy. However, as illustrated in Figure 2.4 full invalidation has a lower opacity cost and uses fewer conflict detection operations as transactions grow in terms of the memory elements they access than validation, resulting in a more efficient TM. Figure 2.4 shows the number of transactions versus the number of operations required for conflict detection, as per the earlier equations.

Our model assumes that all transactions reach their commit phase before conflicts are identified, requiring that each transaction perform incremental opacity (via version-based validation or invalidation's *valid* flag). Figure 2.4 demonstrates the operational overhead of (1) incremental validation (valid), (2) commit-time invalidation using a logarithmic-time search, i.e., $s_{rj}(r_j)$ and $s_{wj}(w_j)$

[5] TL2 has a space complexity optimization for read-only transactions, yet, it requires the transaction be known as read-only before it is executed. In other words, the transaction must be identified as read-only statically, such as at compile time [17].

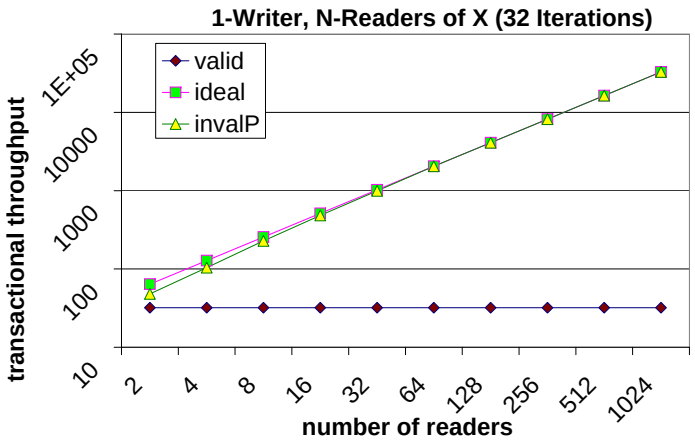


Figure 2.5: Transaction Throughput for 1-Writer / N-Readers of Single Variable.

at $O(log_2 N)$ (invalL), and (3) commit-time invalidation using a constant-time search, i.e. $s_{rj}(r_j)$ and $s_{wj}(w_j)$ at $O(1)$ (invalB). While the commit-time invalidation logarithmic- and constant-time search shows little analytical operational difference in Figure 2.4, their actual executions are noticeably different (see Section 2.3). In Figure 2.4, as the number of elements inserted per transaction increases (from 1, to 10, to 100), the operational delta between validation and invalidation widens by an order of magnitude with each iteration (from 10^2, to 10^3, to 10^4 operational difference). This illustrates our prior point that incremental validation's overhead increases at a disproportionally increased rate when compared to invalidation as transactions access more memory.

2.2.4 An Analysis of Transaction Throughput

We now turn our attention to highly contending, highly concurrent workloads, where concurrency can be exploited but only if the CM can make informed decisions about which transactions to commit and which to abort. We analyze the scenario shown in Figure 2.1, where one transaction writes to X followed by N transactions reading X. We assume lazy write acquisition and that the writer reaches its commit phase first, followed by the N readers. Using this model, Figure 2.5 displays the amount of transaction throughput (y-axis) achieved as N increases (x-axis) using (1) version-based validation (valid), (2) ideal throughput or unfair commit-time invalidation (ideal), and (3) priority-based commit-time invalidation (invalP) [27, 83].

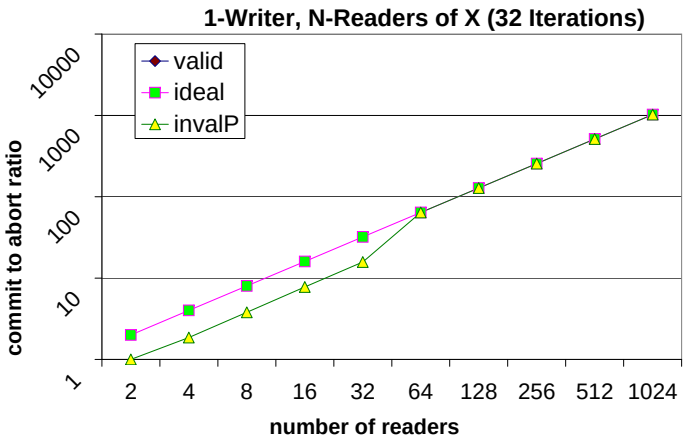


Figure 2.6: Commit to Abort Ratio for 1-Writer / N-Readers of Single Variable.

The priority-based CM policy (3) ensures transactions will not starve, while simultaneously aiming for a high degree of transactional concurrency. It behaves in the following way. Each new transaction begins with a priority of 1. Each time a transaction is aborted, its priority is incremented by 1. Once it commits, the transaction's priority is reset to 1. For a transaction to abort other transactions, its priority must be greater than or equal to the sum of all the transactions it is attempting to abort.

Figure 2.5 shows 32 iterations of the 1-writer / N-readers scenario, where priority-based invalidation eventually achieves $\approx 10^3\times$ greater transaction throughput than version-based validation (for 1024 readers). [6] Figure 2.6 shows the commit to abort ratio of the Figure 2.5. Validation's commit to abort ratio ranges from 0.5 (2 transactions) to 0.000977 (1024 transactions), while priority-based invalidation ranges from 1 (a $2\times$ difference) to 1024 (a $10^6\times$ difference).

2.3 InvalSTM: A Fully Invalidating STM

In this section, we explain the design of InvalSTM, our fully invalidating STM. InvalSTM uses commit-time invalidation for all conflicts (i.e., W-W, W-R, and R-W). InvalSTM also uses lazy write acquisition because it provides the CM with one-to-many conflicts at commit-time.

[6] Our tests of up to 32,768 iterations show that priority-based invalidation retains $\approx 10^3\times$ greater transaction throughput compared to version-based validation, although with each increased N^2 iteration invalidation increases in its transaction throughput by $\approx 1.5\times$.

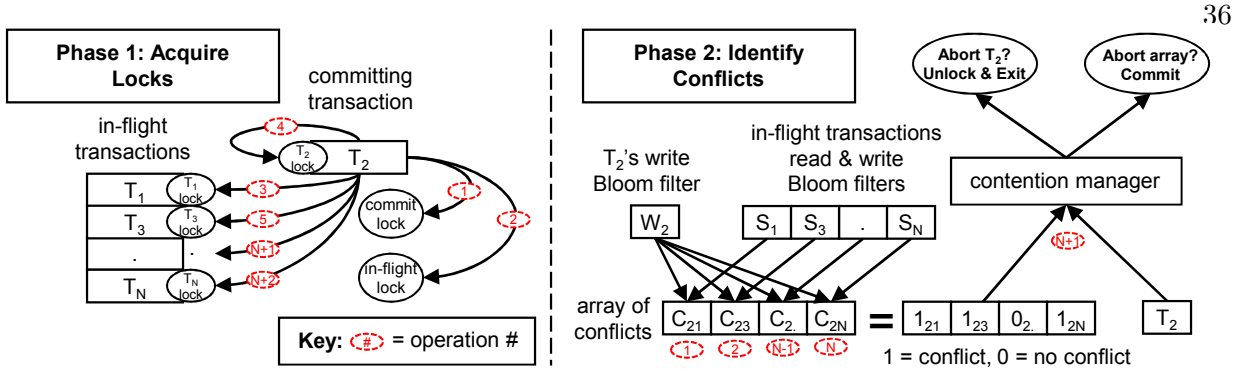


Figure 2.7: An Example of InvalSTM's Commit-Time Invalidation Process.

These one-to-many conflicts are necessary for the CM to make informed decisions that can increase transaction throughput. *Eager write acquisition*, which exclusively acquires write locations as the transaction executes them, sends eager one-to-one conflicts to the CM. These one-to-one conflicts prevent the CM from seeing the entire view of conflicts and, because these conflicts are eager, force the CM to make speculative decisions that can notably limit transaction throughput.

2.3.1 A Design Overview

As explained in Chapter 1, maintaining visible read sets through orecs can be expensive. InvalSTM addresses this in the same way JudoSTM [63], NOrec [14], and RingSTM [86] do, by avoiding orecs altogether. Instead, InvalSTM binds read and write sets directly to a transaction object. In addition, because lock-based STMs have emerged with strong performance – DracoSTM [26], Ennal's STM [21], RingSTM [86], and TL2 [17] – InvalSTM uses mutual exclusion locks as its core synchronization type.

For InvalSTM to perform commit-time invalidation its transactions must be prevented from adding new memory elements to their read and write sets while a transaction commits. Without this restriction, a conflicting memory element may be added to an in-flight transaction's read or write sets after it has been found to be free of conflicts. If such a case were to arise, the total order of transactions may be unserializable. [7]

[7] In fact, only read elements should be prevented from being added to a transaction's read set when another transaction is committing. Preventing write elements from being added to transactions is overly conservative in a

To prevent this unwanted behavior, InvalSTM associates a lock with each transaction. Before performing commit-time invalidation, InvalSTM acquires the transactional locks of all active transactions to ensure the invalidation phase will be performed without extraneous modification to the active transactions' read and write sets.

While this addresses the above concern, it creates a new problem: a serialization point is created from the beginning of a transaction's commit phase until its end. To minimize the negative impact of this serialization point, InvalSTM compresses read and write sets within Bloom filters, which align with caches for improved access time over main memory access time while also providing constant time ($O(1)$) set intersection operational overhead. While storing read and write sets within Bloom filters is not a panacea, it can noticeably reduce the operational overhead of conflict detection. In particular, if alternative data structures were used, such as a linked list or red-black tree for storing read and write sets, these data structures would yield linear ($O(N)$) to logarithmic ($O(N)$) operational overhead when determining if a committing transaction's write element is contained in either the read or write set of an active transaction. The linear or logarithmic overhead would then need to be performed W number of times, where W is the number of write elements in the committing transaction's write set.

The remainder of this section discusses the main design points listed above in greater detail. Below is a list of some of the unique optimizations that emerge from InvalSTM's design.

- Full invalidation is supported, gaining all the benefits highlighted in Section 2.2, including increased transaction throughput resulting from informed CM decisions, zero conflict detection operations for read-only transactions, and efficient conflict detection for memory-intensive transactions.

- Read sets can be stored in imprecise, compressed, and contiguous storage that reduce the time and space complexity to perform invalidation while also reducing cache line eviction rates.

full invalidation TM and is not necessary. We will explain this in more detail in Chapter 3.

- Per-memory locking (orecs) is no longer necessary. Instead, locks are associated with each transaction which can reduce atomic (fenced) operations when transactions are memory-intensive [63, 86].
- Visible read sets using per-transaction storage require zero operations to cleanup, which can reduce serialization when compared to visible read sets using per-memory (orec) storage.

2.3.2 A Lock-Based STM

In addition to a lock per transaction, InvalSTM uses two global locks: a commit and in-flight lock. The commit lock limits the commit phase to a single transaction. The in-flight lock is used to limit modification of the in-flight, or active, transaction list to a single thread.

Before performing commit-time invalidation, the commit lock is acquired to prevent two or more transactions from concurrently committing. While concurrently committing transactions could increase transaction throughput, such behavior could also introduce livelocks. Livelocks could occur because a committing transaction may not be guaranteed to have the time necessary to invalidate all other concurrently committing transactions unless committing transactions are prevented from entering the commit phase until all other committing transactions have completed their invalidation. A concrete example of this is as follows. Consider a committing transaction, T_1, and a second transaction, T_2, entering the commit phase. T_1 invalidates T_2. However, before T_1 can successfully finish its commit operations, T_2 re-enters the commit phase and invalidates T_1. This scenario could repeat indefinitely creating a perpetual cycle of invalidation where the system is livelocked on T_1 and T_2. This livelock cycle can decrease or even halt throughput, so InvalSTM prohibits it by limiting the commit phase to a single transaction.

The in-flight lock is acquired before commit-time invalidation is performed so new transactions cannot be put in-flight. This is needed for two reasons. First, it creates a sequential locking order based on the transactions that are currently in-flight (as seen in Figure2.7). Second, it prevents a livelock that could occur as invalidated transactions are removed and placed back in-flight,

requiring cyclic invalidation (a variant of the above livelock commit-time behavior with T_1 and T_2).

While these additional locks complicate the design, their absence would reduce concurrency in the following ways. First, if only the commit lock was used, all transactions would be required to obtain it when adding elements to their read and write sets. This would effectively serialize all read and write operations between transactions, even non-conflicting ones. By using the commit and transactional locks, transactions can concurrently add elements to their read and write sets, so long as no transaction is committing. Second, if the in-flight lock was removed and instead the commit lock was used to add or remove transactions from the in-flight set, the system could only either allow a transaction to commit or begin, both operations could not happen concurrently. While it is true that committing transactions generally do need to obtain the in-flight lock, they do not always need it immediately nor do they need it for the entire duration of the commit phase. Transactions whose $valid = false$ can perform nearly all of their cleanup code prior to requiring the in-flight lock. In addition, read-only transactions only require the commit lock momentarily to identify that they are in fact read-only and to check that they are *valid*. Once checked, these transactions release the commit lock, but retain the in-flight lock to remove themselves from the list. Lastly, transactions who have written their local write data to the globally shared location may have additional cleanup operations to perform that can require the commit lock but can be done without holding the in-flight lock, such as commit phase bookkeeping.

2.3.3 Serialized Commit

A downside of InvalSTM's locking design is that it creates a serialization point during a transaction's commit phase. This serialization point limits commits to one transaction at a time and prevents in-flight transactions from adding new elements to their read and write sets while a transaction commits. To minimize the negative impact of this serialization point, read and write sets are stored in Bloom filters which can reduce the total execution time of the invalidation process by performing set intersection in constant worst-case time [9]. Below is the modified operational overhead equation ($o_{ib}(M)$ for M transactions) from Section2.2 when read and write sets use Bloom

filters to perform full invalidation.

$$o_{ib}(M) = \sum_{i=1}^{M} r_i + (2kw * (F_i))$$

Commit-time invalidation is handled by $\sum_{i=1}^{M} 2kw * (F_i)$ where w is the number of words per bit vector, k is the number of bit vectors per Bloom filter, and F_i are the in-flight transactions at the time the ith transaction is committing. The opacity checks, which are performed throughout the transaction's lifetime, add r_i (number of elements in the transaction's read set) to each summation. By using Bloom filters, the original search operations required for a single transaction is reduced from w_i to 1 (Section 2.2), because the conflicts between one transaction and another are found in a single set intersection. kw represents the original equation's search time of $s(w_j)$ and $s(r_j)$. Since kw must be performed twice (once per read and write set) the result is $2kw$. Because these operations must be performed for each in-flight transaction, we multiply $2kw$ by F_i.

For each transaction, InvalSTM currently uses a fixed 2^{16} bits per bit vector and two bit vectors per Bloom filter. Although we experimented with a wide variety of Bloom filter configurations, our early experiments indicate the current size performs the best overall for our tested benchmarks. We expect to extend our research in this area as we analyze more benchmarks.

2.3.4 Transaction Implementation

In InvalSTM each transaction object contains its own read and write sets. Read sets store memory locations, while write sets store memory locations plus a copy that is used to buffer transactional writes for lazy write acquisition. The memory locations for read and write sets are stored in separate Bloom filters. This is done so different types of conflicts can be handled by the CM in different ways. For example, if a committing, writer transaction has a W-R conflict with another active reader, the CM can choose to stall the writer until the reader transaction commits. If read and write sets were not separated, the CM may only be able to resolve this conflict via an abort.

Because write sets store written data along with memory locations, each transaction contains an additional write map that associates written data with its memory location. This data structure

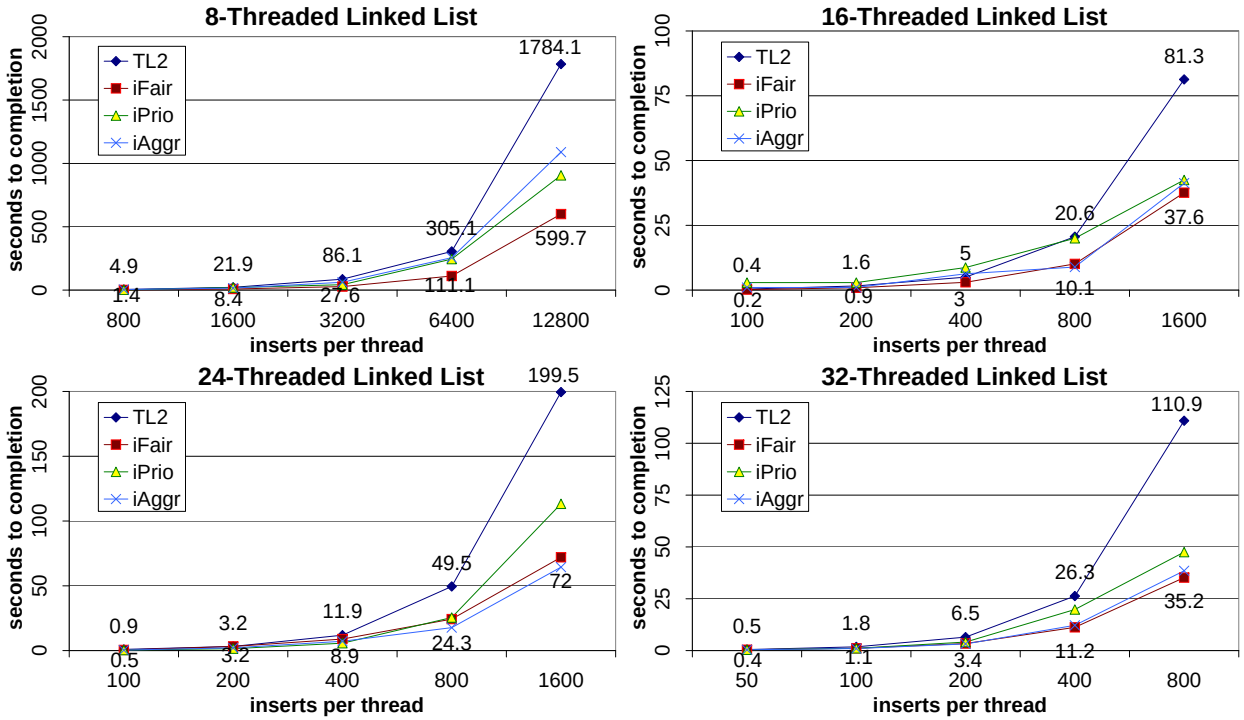


Figure 2.8: Linked List Benchmarks.

is necessary in addition to the Bloom filter used for write sets, because lazy write acquisition must commit memory in such a way that false positives are not possible [9]. Otherwise the TM could not perform reliable updates to specific memory locations.

In a fully invalidating TM, ensuring a transaction is opaque is inexpensive (an $O(1)$ operation). Therefore, InvalSTM performs opacity checks on all transaction calls, not just the ones in which it is necessary. This adds some overhead, yet, we have found it improves system performance because it can identify doomed transactions earlier than what would be possible if opacity checks were only performed when new memory elements were opened for reading.

When a transaction accesses a memory element for reading or writing, InvalSTM performs a read-only lookup to see if the transaction has already accessed the element. This lookup requires no locking, because the operation is not changing the read or write sets. If the lookup is successful, the appropriate value is returned. If not, the transaction's lock is acquired, the memory address (and value if necessary) is inserted into the correct set, and the transaction's lock is released.

A benefit of this approach is that once a memory element is accessed, as long as it is accessed in the future with the same or reduced strength (i.e., a read access remains a read access, or a write access is reduced in strength to a read access), the accessing transaction is never blocked regardless of if another transaction is committing. This is because the committing transaction already has access to the memory element because it has been accessed in the past and the in-flight transaction is only required to obtain its transactional lock when a new element is added to its read or write set.

Figure 2.7 provides a high-level view of the commit-time invalidation process. For brevity, some details are omitted from the diagram. Those include the priority elevation for aborted transactions, the removal of aborted transactions from the in-flight set to reduce in-flight lock contention, the CM's usage of transaction size (read + write sets) as a discriminator for the abort protocol, and the short-circuited logic used for read-only transactions.

The commit-time invalidation process, shown in Figure 2.7, begins with Phase I where the commit and in-flight locks are acquired. This ensures no other transaction can commit or be started while a transaction is committing. Next, the committing and in-flight transactions' associated locks are acquired in a sequential order to avoid deadlock. Phase II then identifies the conflicts the committing transaction has with the in-flight transactions. If conflicts exist, the CM is sent the batch of conflicts. The CM then makes a decision on which transactions should be aborted or stalled based on the information that it has been supplied.

An important detail of InvalSTM's design is that in-flight transactions can make forward progress during the entire commit-time invalidation process. The two exceptions are (1) transactions cannot concurrently commit while another transaction is already committing and (2) they cannot add new memory elements to their read and write sets. Those limitations aside, some of our early experiments, which are not presented here, have shown that allowing active transactions to make forward progress while another transaction is committing does in fact increase overall throughput for workloads that access a shared memory element multiple times (e.g., a head or sentinel node, a global counter, etc.).

As Bloom filters can emit false positives there exists the possibility that these false positives can prevent forward progress. To avoid this scenario, we use a run-time threshold R that, once exceeded by our abort to commit ratio, switches our TM from using Bloom filters to using red-black trees for read and write sets, ensuring false positives are avoided. After some programmable period of time T, our system reverts back to using Bloom filters for read and write sets. In our experiments, however, this threshold is never reached.

2.3.5 Experimental Results

In this section we present the experimental results comparing InvalSTM, using the commit-time invalidation design explained above, against TL2, a state-of-the-art validating STM. The benchmarks were run on a 1.0 GHz Sun Fire T2000 supporting 32 concurrent hardware threads with 32 GB RAM. The TL2 implementation used for these experiments is from RSTM.v4, University of Rochester's STM library collection. For all the graphs in this section, the y-axis shows the total execution time in seconds (lower is always better). The x-axis represents the workload executed rather than the usual threads, because, as shown in Section2.2, invalidation performs more efficiently when compared to validation when transactions access more memory. Because the number of threads is constant per graph, four graphs are used per benchmark each with a different thread count and/or workload configuration.

-

2.3.5.1 Contention Manager Variants

We tested three CM variants with our benchmarks: iFair (invalidation fair), iPrio (invalidation prioritized) and iAggr (invalidation aggressive). iAggr ensures the first transaction to enter the commit phase commits. It demonstrates how commit-time invalidation performs when it does not use conflict information to make informed decisions. In other words, iAggr specifically captures performance difference, in terms of execution time, between invalidation and validation's conflict detection and opacity operational difference.

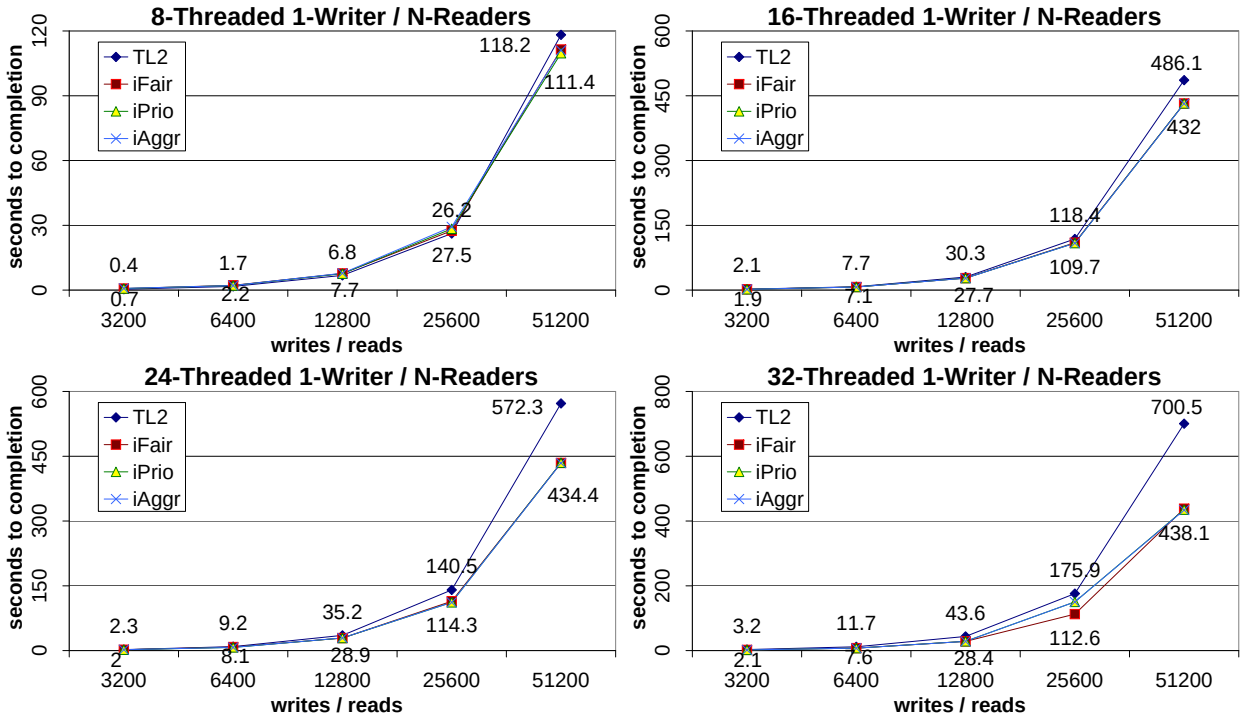


Figure 2.9: 1-Writer, N-Reader Benchmarks.

iPrio associates a priority with each transaction. A transaction's priority is raised each time it aborts and is reset each time it commits. A transaction can commit if it has the highest priority of all conflicting in-flight transactions. A transaction can also commit if it has the largest read set size of all in-flight transactions or its read and write set size is larger than the average read and write set size of all conflicting transactions plus their cumulative priority.

iFair associates a priority with each transaction and raises and resets the transaction's priority in the same manner as iPrio. Unlike iPrio, a transaction can commit if its read and write set size is greater than a weighted average of the in-flight transactions' read size and priority. A transaction can also commit if its read and write set size is greater than any of the in-flight transactions' read set size. In addition, if an in-flight transaction's read set size is $10^2\times$ greater than the committing transaction's read set size and its priority is $2^3\times$ greater, iFair will abort the committing transaction in favor of the higher priority, larger in-flight transaction.

Of the three CM variants, iFair performs the best overall. While iAggr and iPrio each perform

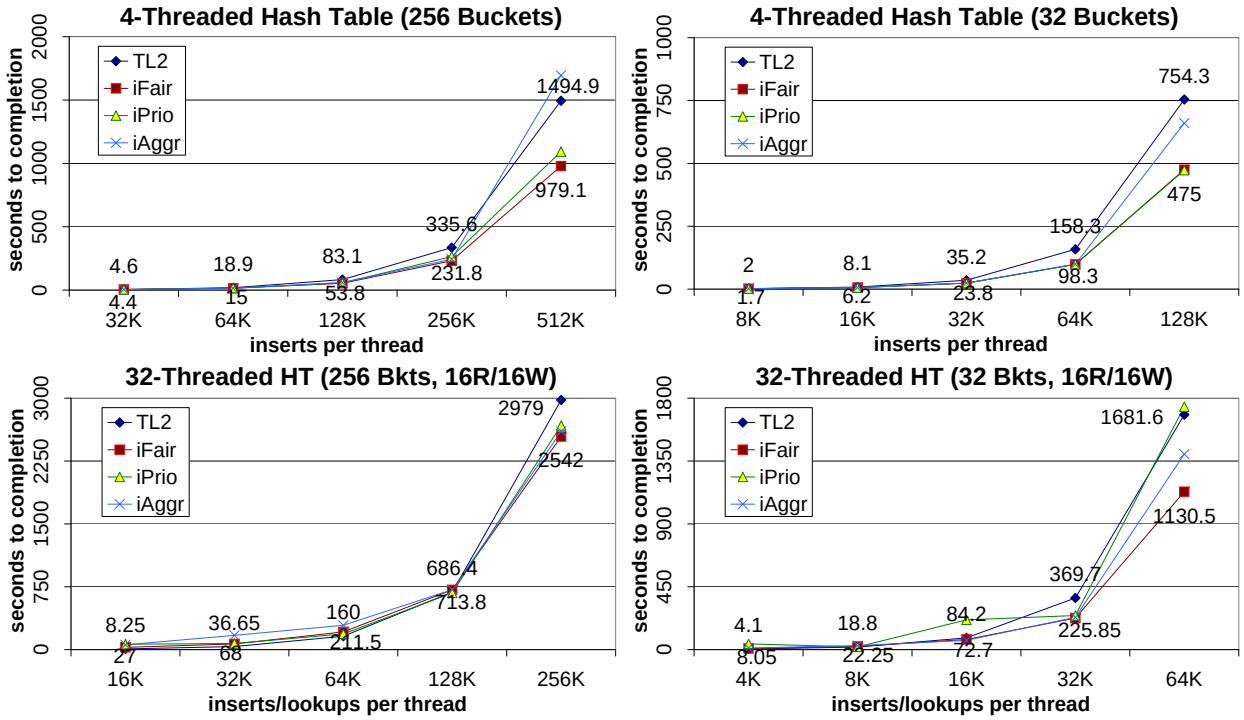


Figure 2.10: Hash Table Benchmarks.

well under certain conditions, iFair consistently performs as well or better than the other two CM strategies. Additionally, in some cases, iFair outperforms TL2 by more than 3× (Figure 2.8, 32-Threaded Linked List). Based on the performance improvement as the concurrency widens from 8 to 32 threads, one might speculate that commit-time invalidation's performance will improve compared to TL2 as the number of concurrent transactions grow.

2.3.5.2 Linked List

Our linked list benchmarks are shown in Figure 2.8. Each linked list benchmark populated a single linked list with N concurrently executing threads. Each thread inserted the same number of elements (i.e., T_1 inserts 0-99, T_2 inserts 100-199, etc.) and the insert operation was a transaction. iFair performed most consistently, especially in the 8-threaded benchmark where its CM policy drives it to outperform the other CM policies by $\approx 2\times$.

In the linked list benchmarks, InvalSTM outperforms TL2 from $\approx 2\times$ to $\approx 3\times$. At nearly

all data points, as the workload increases InvalSTM improves its efficiency over TL2. For the final data point in the 32-threaded benchmark, InvalSTM's iFair is 3.15× more efficient than TL2. It is important to note that the larger threaded benchmarks perform less work than the smaller threaded benchmarks (e.g., the 32-threaded workload inserts <= 800 nodes per transaction, while the 8-threaded one inserts <= 12,800 nodes). However, the performance difference for InvalSTM and TL2 is roughly the same for all threaded executions. This suggests that if equivalent work was executed for the 32-threaded benchmarks, the performance difference would significantly favor InvalSTM.

2.3.5.3 1-Writer / N-Readers

For highly contending but also highly concurrent workloads commit-time invalidation performs well. This is demonstrated in the 1-writer / N-reader benchmark shown in Figure 2.9. The 1-writer / N-reader benchmark was implemented using a linked list where the writer performs a fixed number of appends to the linked list and each reader performs a fixed number of lookups in the list, where the lookup value is increased by one with each lookup iteration. Both the append and lookup operations are transactions. While the performance difference between InvalSTM's CM strategies and TL2 for 8 and 16 threaded workloads is relatively negligible, the performance difference between the 24 and 32 threaded workloads is more pronounced. The 32 threaded benchmark shows that iFair outperforms TL2 by $\approx 1.6\times$. The reason for this is straightforward: lower threaded workloads emit fewer aborts because contention on the data is reduced. As readers are added the contention increases as do the number of aborts. This creates a scenario where the early notification of doomed transactions, a low overhead benefit of invalidation, and the reduced operational overhead and serialization to perform opacity checks, which are compounded by concurrently executing transactions, are responsible for reducing a significant number of latency-related computational operations. By reducing the total number of these operations by a notable amount (as discussed in Section 2.2), the total system execution time is also reduced thereby improving system performance.

While the 1-writer / N-reader performance difference favors InvalSTM by only $\approx 1.6\times$, this margin is notable because TL2 has a space and time optimization for read-only transactions, known as early release [40], though transactions must be flagged as read-only prior to executing. Commit-time invalidation has a time optimization which can be used for both static and dynamic read-only transactions. However, to be fair to TL2, we only leverage read-only optimizations for statically tagged read-only transactions, because we felt it would be unfair to exploit an invalidation optimization that did not have a counterpart in validation.

Because N of the transactions are read-only (where N = 7, 15, 23, and 31), both systems heavily exploit their read-only transaction optimizations for this benchmark. Although TL2's read-only optimizations are impressive, InvalSTM's read-only optimizations seem to have more impact on performance for this particular scenario.

2.3.5.4 Hash Tables

The hash table experiments are shown in Figure 2.10 and are implemented using N-bucketed lists (N = 32 and 256). The hash function is a modulo operation on the number of buckets. Each benchmark used a single hash table which was concurrently populated by N number of threads and used the same insert conditions as the linked list example.

For the smaller sized hash tables, TL2 is upwards of $3\times$ faster than InvalSTM. However, as the hash table sizes are increased, InvalSTM eventually performs more efficiently than TL2 by $1.48\times$ and $1.58\times$ for the 256 and 32 bucketed hash tables, respectively. For the 32-threaded 16 reader / 16 writer benchmarks, InvalSTM is $1.17\times$ faster than TL2 for the 256 bucketed hash table. InvalSTM is $1.48\times$ faster than TL2 for the 32 bucketed one.

Although these performance improvements for InvalSTM are lower than the linked list and the 1-writer / N-reader experiments, they are meaningful because a bucketed hash table is generally considered to be a data structure that is inherently concurrent. As such, a hash table's operations are distributed across numerous and simultaneously accessible buckets [51]. Due to this, one might suppose that invalidation would perform poorly because this structure does not inherently have

the same contentious behavior as the prior benchmarks. However, after the hash table buckets reach a certain threshold of size, the transactions inserting on the buckets begin to contend since appending elements to densely populated buckets requires more transactional execution time and will eventually lead to multiply contending transactions on the same bucket. Notice that even for the 32-threaded hash table benchmarks, InvalSTM outperforms TL2 by up to $\approx 50\%$.

2.4 DracoSTM

DracoSTM is a lock-based STM that implements full invalidation. At its core, and much like InvalSTM, DracoSTM uses one lock per thread that is acquired each time a transaction opens a new element for reading or writing. This allows multiple transactions to simultaneously read and write new data without blocking other transactions' progress. When a transaction is committing, the same global locking strategy as used in InvalSTM, is used in DracoSTM. This locking design temporarily blocks forward progress on all transactions except the committing one. Once the committing transaction completes, other transactions are allowed to resume their work. Transactions that are not adding new memory elements to their read or write sets can continue to make forward progress while a transaction is committing. More details of DracoSTM can be found in [24]. In the above ways, DracoSTM is similar to InvalSTM.

In other ways, DracoSTM is different from InvalSTM. Instead of Bloom filters for read and write sets, DracoSTM uses hash tables to store read and write sets which incur roughly $O(log_2(N))$ search time complexity as opposed to the constant set intersection time of Bloom filters. DracoSTM also supports both direct and deferred update, while InvalSTM only supports deferred update. Furthermore, DracoSTM supports both types of update dynamically, meaning DracoSTM can switch between direct and deferred update at run-time. However, because DracoSTM supports run-time alternation between updating strategies, the system must have mechanisms in place that switch to the appropriate updating policy based on which one is currently active.

Furthermore, because DracoSTM uses direct update, DracoSTM must support eager W-W conflict detection. Lazy W-W conflict detection is not possible when using direct update because

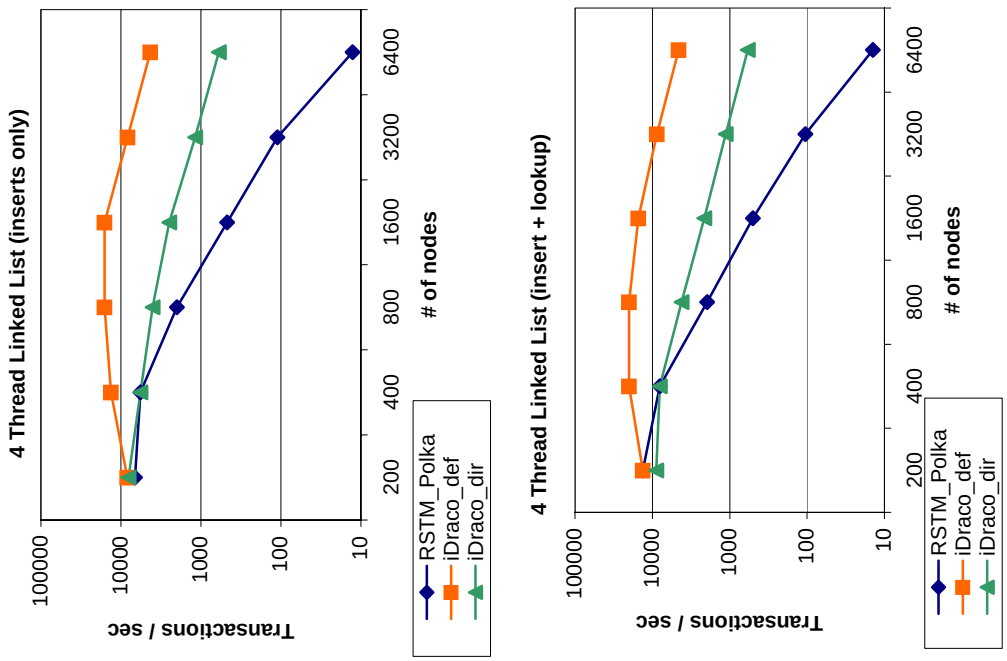


Figure 2.11: Four Threaded Linked List Benchmark.

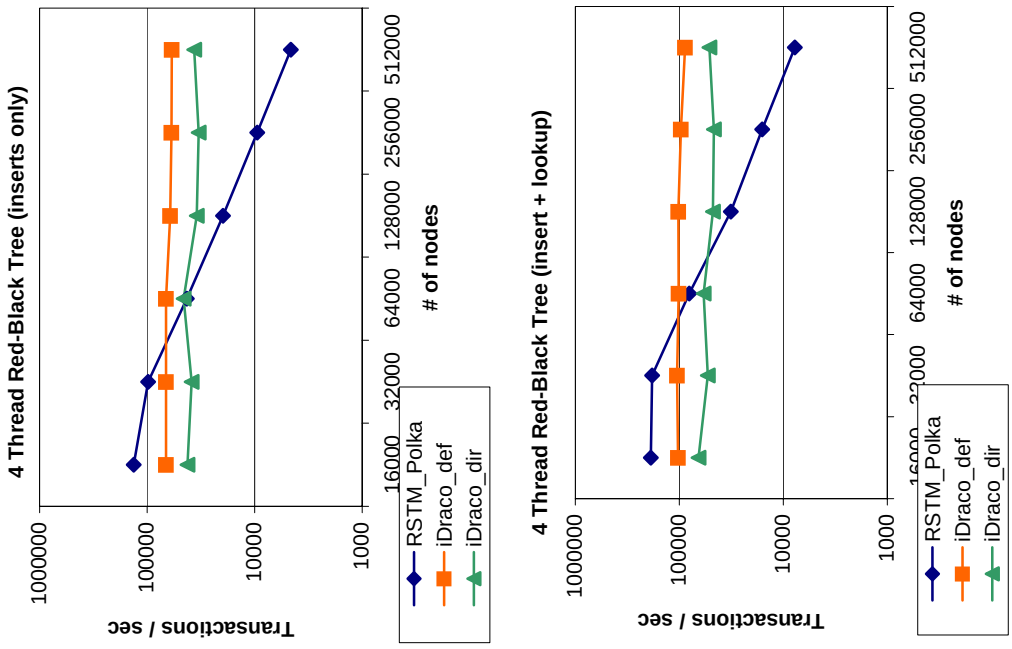


Figure 2.12: Four Threaded Red-Black Tree Benchmark.

when multiple, active transactions eagerly write to the same shared data location the TM may not be able to provide some form of serializability between the concurrent writers. Because of this, DracoSTM's direct update CM, or direct update's CM in general, cannot handle bulk conflict detection. Instead, it is relegated to one-to-one conflict detection for W-W conflicts, which can limit the information the CM receives regarding conflicts and relies on the CM to make some speculation about the future events of transactions.

Because DracoSTM must support eager W-W conflict detection and resolution, it also allows the programmer to decide when to perform W-R and R-W conflict detection; the programmer can perform W-R and R-W conflict eagerly, at conflict-time, or lazily, at commit-time. InvalSTM can only perform conflict detection at commit-time.

2.4.1 Experimental Results

The experimental results shown in Figures 2.11 and 2.12 were gathered in early 2007, when non-blocking STMs were still the primary way in which to implement STMs, and RSTM was one of the best performing non-blocking STMs.

Figure 2.11 and 2.12 illustrate DracoSTM's performance for four threaded linked list and red-black trees. These performance operations compare RSTM, University of Rochester's non-blocking C++ STM library, to DracoSTM using deferred and direct update. Both Figure 2.11 and 2.12 double the number of nodes inserted and looked up with each iteration. The x-axis shows the number of nodes inserted (and looked up) in the container and the y-axis shows the number of transactions per second. The higher the transactions per second, the faster the STM system is performing. These tests were run on a 3.2 GHz 4-processor Intel Xeon with 16 GB RAM.

While RSTM is a fast performing non-blocking STM, the benchmarks show that DracoSTM is more efficient than RSTM in many cases. On the other hand, the benchmarks also show that RSTM is as efficient or more efficient than DracoSTM when transactions that access few memory elements dominate the workload as can be seen in the left-most points of both Figure 2.11 and Figure 2.12. Figure 2.11 and 2.12 demonstrate the superlinear performance improvement DracoSTM has over

RSTM as the size of the container doubles. For example, the 6400 linked list insert and lookup test took RSTM 908 seconds to perform while DracoSTM performed the same workload in 2 seconds using deferred update (454x faster) and in 22 seconds using direct update (41x faster). The 512,000 red-black tree insert + lookup test took RSTM 132 seconds to perform, while DracoSTM performed the same operation in 10 seconds using deferred update (13x faster) and 19 seconds using direct update (7x faster). We believe these performance metrics are partially the result of library optimizations found within DracoSTM, but are mostly due to specific TM characteristics of the system implementation (e.g., lock-based deferred update using commit-time invalidation).

Chapter 3

The Theory of Full Invalidation

To address the need for improved efficiency and to support an ever-increasing feature set, the state-of-the-art TM systems have evolved in such a way that they are vastly different than the pioneering systems from a decade ago. An unfortunate side-effect of this evolution is that some of the early theoretical work on TM correctness no longer applies to these new state-of-the-art systems. Because of this, it is difficult to prove that full invalidation is correct in terms of conflict and view serializability using the existing theoretical TM models.

In this chapter, we present a new theoretical model for TMs that use deferred update and use it to prove certain correctness characteristics for full invalidation. The overarching goal of this chapter is to provide a theoretical treatment to full invalidation and to demonstrate that the systems that implement some form of full invalidation, such as InvalSTM, meet some minimum criteria of correctness. In this case, that minimum criteria of correctness are conflict and view serializability, which will be defined shortly.

Recall that full invalidation is a conflict detection strategy where a committing transaction resolves its conflicts with active transactions before it commits [33]. This differs from validation in that validation checks a committing transaction for conflicts with transactions that have already committed. Because of the way in which validation identifies conflicts, a validating TM must abort all committing transactions if conflicts are found between it and other transactions that would lead to an unserializable history. With full invalidation, if a conflict is found between a committing transaction and other active transactions, the TM always has a choice: it can (1) stall or abort the

committing transaction or (2) it can abort the conflicting, active transactions. Making informed decisions regarding these choices allows a full invalidation TM to be more efficient than validation in terms of transaction throughput.

In the upcoming sections, we formally define full invalidation, originally proposed by Gottschlich et al. [33], in terms of a TM automaton and provide the definitions and proofs that are necessary to show that full invalidation is conflict and view serializable. Our approach uses a model adapted from Lynch et al.'s I/O automata [57]. We primarily use view serializability as our criterion of correctness as defined by Papadimitriou [68]. We extend Papadimitriou's definition of a conflict graph to include a new type of graph, which we call a *lazy conflict graph*, that is specifically designed for deferred update TMs. A benefit of using a lazy conflict graph is that it provides a more relaxed system in which a TM can accept a wider range of deferred update concurrent histories than Papadimitriou's original conflict graph. Our model is described in the following sections. Some of the definitions are extensions or modifications of definitions by Lynch et al. [57], Papadimitriou [68], and Ramadan et. al [74].

3.1 Preliminary Definitions

A *history*, or *schedule* [68], is a sequence of instantaneous *events*. For our automaton, events are read, commit, or abort, as defined below.

- $\langle T, x.read(v)\rangle$: Transaction T reads value v from variable x.
- $\langle T\ commit, I, W\rangle$: Transaction T commits, invalidating a set of transactions I and atomically writing W, a function mapping variables to values.
- $\langle T\ abort\rangle$: Transaction T aborts.

If h is a history and S a set of transactions, the *projection* of h onto S, denoted by $h|S$, is the sub-sequence of events in h for all transactions in S [57, 74].

An event e_1 is said to *happen before* e_2 in history h, denoted by $e_1 <_h e_2$, if e_1 occurs in h before e_2 occurs in h [53]. We define a *conflict* between two events e_1 and e_2 in history h, denoted

as $e_1 \prec\!\succ_h e_2$, as:

$$e_1 \prec\!\succ_h e_2 \text{ iff } \textit{conflicts}-\textit{with}(e_1, e_2, h) \text{ or } \textit{conflicts}-\textit{with}(e_2, e_1, h)$$

where

$$\textit{conflicts}-\textit{with}(e_1, e_2, h) \text{ iff } WW(e_1, e_2, h) \text{ or } WR(e_1, e_2, h) \text{ or } RW(e_1, e_2, h)$$

and where WW, WR, and RW are defined as:

- $WW(e_1, e_2, h)$ iff $e_1 = \langle T_1\ \textit{commit}, I_1, W_1\rangle \in h$, $e_2 = \langle T_2\ \textit{commit}, I_2, W_2\rangle \in h$, $\textit{dom}(W_1) \cap \textit{dom}(W_2) \neq \emptyset$, and $e_1 <_h e_2$ for some T_1, T_2, I_1, I_2, W_1, and W_2.
- $WR(e_1, e_2, h)$ iff $e_1 = \langle T_1\ \textit{commit}, I_1, W_1\rangle \in h$, $e_2 = \langle T_2, x.\textit{read}(v)\rangle \in h$, $x \in \textit{dom}(W_1)$, and $e_1 <_h e_2$ for some T_1, T_2, I_1, I_2, W_1, and W_2.
- $RW(e_1, e_2, h)$ iff $e_1 = \langle T_1, x.\textit{read}(v)\rangle \in h$, $e_2 = \langle T_2\ \textit{commit}, I_2, W_2\rangle \in h$, $x \in \textit{dom}(W_2)$, and $e_1 <_h e_2$ for some T_1, T_2, I_1, I_2, W_1, and W_2.

We lift the notion of conflict to transactions with the following definitions. There is a *conflict* between two transactions T_1 and T_2 in history h, denoted as $T_1 \prec\!\succ_h T_2$, iff $e_1 \in h|T_1$ and $e_2 \in h|T_2$ and $e_1 \prec\!\succ_h e_2$ for some e_1 and e_2. We lift $\textit{conflicts}-\textit{with}$ to transactions in a similar fashion. $\textit{conflicts}-\textit{with}(T_1, T_2, h)$ iff $e_1 \in h|T_1$ and $e_2 \in h|T_2$ and $\textit{conflicts}-\textit{with}(e_1, e_2, h)$ for some e_1 and e_2. Likewise for WW, WR, and RW.

Two histories are conflict equivalent if both histories contain the same transactions and if each pair of conflicting events has the same happens before ordering in both histories. More formally, a history h_1 is *conflict equivalent* to a history h_2, denoted by $h_1 \equiv_{ce} h_2$, when h_2 is a permutation of h_1 and for every $e_1, e_2 \in h_1$, if $e_1 \prec\!\succ_{h_1} e_2$ then $(e_1 <_{h1} e_2)$ *iff* $(e_1 <_{h2} e_2)$. A history is *serial* if it consists of a succession of transactions, without interleaving any events from distinct transactions [68]. For a serial history h, a transaction T_1 is said to *happen before* T_2, denoted by $T_1 <_h T_2$, if and only if all of the events of $h|T_1$ precede all of the events of $h|T_2$.

A history is *conflict serializable* if it is conflict equivalent to a serial history. If h is a history, T is a transaction, and x is a variable, then:

- $active(h)$ is the set of active (not committed or aborted) transactions.
- $valid(T_1, h)$ iff $\forall T_2, I, W.$ if $\langle T_2\ commit, I, W\rangle \in h$ then $T_1 \notin I$.
- $value(x, h)$ returns the associated value v from the last commit event, that is, $\langle T\ commit, I, W\rangle$ is the last commit event and $(x, v) \in W$.
- $writes(T, h)$ is the set of written variables ($dom(W)$) of T's commit event in history h.
- $reads(T, h)$ is the set of variables in read events by T in history h.
- $length(h)$ is the number of events in h.
- $allowAbort(T_1, T_2, h)$ is true if, in history h, transaction T_1 is given permission to abort transaction T_2 by the contention manager, otherwise it is false.
- $inval(h)$ is true if h is accepted by a full invalidation TM automaton (as described in Figure 3.1), otherwise it is false.
- $conflict\_serial(h)$ is true if h is conflict serializable, otherwise it is false.

3.2 Full Invalidation TM Automaton

A full invalidation TM automaton is a concurrency control mechanism that accepts (or rejects) a history. The automaton has an associated history, h, where $h = \emptyset$ before the automaton accepts any event. When an event e_i is accepted by the automaton, a corresponding event e_o is appended to the automaton's history, so $h' = h \cdot e_o$, where h represents the history prior to processing e_i and h' represents the history after processing it. A full invalidation TM automaton is defined by its preconditions and postconditions for the events it accepts, also known as its *transition relation* table [57], which is formally presented in Figure 3.1.

The intuition behind the design of the full invalidation TM automaton is as follows. First, full invalidation requires that each committing transaction identify and resolve conflicts that exist between it and all active transactions before it commits. The Commit.1 action captures the

situation in which the committing transaction is given permission to commit by the automaton. The Commit.2 action captures the situation in which the committing transaction is aborted by the automaton.

When a conflict is detected, the contention manager decides if the committing transaction is allowed to tell the other active transactions that it conflicts with to abort (as represented by the check for *allowAbort*()). The two commit actions, Commit.1 and Commit.2, handle the two outputs, true or false, that *allowAbort*() can return.

In our implementation of InvalSTM, when a committing transaction is given permission to tell the other conflicting transactions to abort, the *valid* flag of those transactions is set to *false*. In the full invalidation automaton, this behavior is captured by recording the invalidated transactions to the I set of the commit operation of the committing transaction: $\langle T\ commit, I, W\rangle$. Then, when these invalidated transactions perform their next transactional operation, they are notified that they have been invalidated by querying the *valid*() operation. If the result is *false*, the full invalidation TM automaton's behavior mirrors the behavior of a system that implements full invalidation, such as InvalSTM, which is to immediately abort the invalidated transaction. These behaviors can be seen more precisely in the formal definition of the full invalidation TM automaton shown in Figure 3.1.

3.3 Graph Representations of Conflicts and Dependencies

The following directed graphs, D, D_2, and G, are used throughout the proofs in Section 3.4.

The D graph captures where read events get their values. The vertices in the graph are events, and each edge in the graph connects a read for some variable to the last commit that wrote the same variable. We also include two extra transactions in a D graph, one at the beginning of the history to initialize all variables and another at the end of the history that reads all variables.

The D_2 graph captures which events conflict with one another and, of these conflicting events, which events must come before other events. The vertices of a D_2 graph are events (read and commit) and the edges record both read-after-write and write-after-read conflicts between events.

- Read

  Pre: $T \in active(h) \wedge valid(T, h)$

  Post: $h' = h \cdot \langle T, x.read(value(x, h))\rangle$

- Commit.1

  Pre: $T \in active(h) \wedge valid(T, h) \wedge (\forall T_x. T_x \in active(h) \wedge valid(T_x, h) \wedge (writes(T, h) \cap reads(T_x, h) \neq \emptyset) \rightarrow allowAbort(T, T_x, h))$

  Post:

  (1) $I = \emptyset$

  (2) $T \in active(h) \wedge valid(T, h) \wedge (\forall T_x. T_x \in active(h) \wedge valid(T_x, h) \wedge (writes(T, h) \cap reads(T_x, h) \neq \emptyset) \rightarrow I = I \cup T_x))$

  (3) $h' = h \cdot \langle T\ commit, I, W\rangle$

- Commit.2

  Pre: $T \in active(h) \wedge valid(T, h) \wedge (\exists T_x.T_x \in active(h) \wedge valid(T_x, h) \wedge (writes(T, h) \cap reads(T_x, h) \neq \emptyset) \wedge \neg allowAbort(T, T_x, h))$

  Post: $h' = h \cdot \langle T\ abort\rangle$

- Abort

  Pre: $T \in active(h)$

  Post: $h' = h \cdot \langle T\ abort\rangle$

Figure 3.1: Full Invalidation TM Automaton.

The G graph lifts the D_2 graph from events to transactions. So all the vertices associated with a transaction in D_2 are represented by a single vertex in G. With that compression, all of the edges in D_2 between events from two transactions are collapsed to a single edge between those two transactions in G. To recap, the G graph captures which transactions conflict with one another and which transactions must come before others.

3.3.1 Formal Definitions of the Graphs

An *augmented history* of h, denoted by $\hat{h}$, is defined as the history h including two additional transactions, T_0 and T_∞. T_0 initializes (writes to) all of the variables in h to some arbitrary value and executes serially before any other transaction in h begins. T_∞ reads all variables in h and executes serially after all other transactions in h have completed.

With augmented histories defined, we can now define the directed graph $D(h)$. $D(h)$'s vertices are the events of the transactions in the augmented history, $\hat{h}$. The edges of $D(h)$ are specified as follows.

- If e_i and e_j are events in different transactions, where e_i is a commit event and e_j is a read event and e_i writes to variable x (i.e., $x \in e_i$'s $dom(W)$) while e_j reads x, and e_i is the last step in $\hat{h}$ before e_j that writes to x, then edge (e_i, e_j) is in $D(h)$.

We now define $D_2(h)$ as a directed graph whose vertices are the events of h and whose edges specify the happens before ordering between conflicting event pairs from different transactions in h. Specifically, there is an edge $(e_1, e_2) \in D_2(h)$ if $conflicts{-}with(e_1, e_2, h)$.

Finally, we define a *lazy conflict graph* $G(h)$ as a directed graph whose vertices are transactions and whose edges specify the happens before ordering between conflicting transactions. There is an edge $(T_1, T_2) \in G(h)$ if $conflicts{-}with(T_1, T_2, h)$.

A lazy conflict graph is named as such because it captures the happens before ordering that exists between transactions in a lazy write acquisition (also known as deferred update) TM that uses an atomic commit operation. In such systems, because writes events always occur during

the commit phase, and because the commit phase is atomic (no other transactions can commit at the same time), the edges formed by the happens before ordering for write operations between transactions are always one directional. In other words, when two transactions have write-after-write conflicts with each other, one transaction is clearly defined as happening before the other. This same behavior is not true in eager write acquisition (also known as direct update) TMs.

Like Papadimitriou's conflict graph [68], a history is conflict serializable *iff* its lazy conflict graph is acyclic (proof forthcoming).

3.3.2 View Serializability

In addition to conflict serializability, there is another form of serializability called *view serializability*. Before we provide the formal definition of view serializability, we first give the intuition behind it. In essence, view serialization is a form of serialization that ensures transactions see a consistent view of the world. For example, using view serializability, an active transaction's operations are required to be isolated from other active transactions. Thus a transaction's *view* of the world is always consistent.

Both view serializability and conflict serializability ensure transactions see a consistent view of the world. However, it is more computationally expensive to verify if a history is view serializable than if it is conflict serializable. Specifically, verifying if a history is view serializable has a complexity of NP, while verifying if a history is conflict serializable has a complexity of P [68]. Although view serializability is more computationally expensive than conflict serializability, view serializability is more relaxed than conflict serializability and therefore accepts a wider range of histories. To speak more formally, in the universe of all histories those histories accepted by a system that accepts all view serializable histories is a superset of those histories accepted by a system that accepts all conflict serializable histories.

In the following section, D graphs are used to prove that if two histories are view equivalent then their respective D graphs are also equivalent. D graphs are also used, indirectly, to prove that if two histories are conflict equivalent then their respective D_2 graphs are equivalent. Two histories

h_1 and h_2 are said to be *view equivalent* when the following conditions are satisfied:

- If the transaction T_i in h_1 reads an initial value (from T_0) for a variable x, so does the transaction T_i in h_2.
- If the transaction T_i in h_1 reads the value written by transaction T_j in h_1 for a variable x, so does the transaction T_i in h_2.
- If the transaction T_i in h_1 is the final transaction to write the value for a variable x, so is the transaction T_i in h_2.

A history h is *view serializable* if it is view equivalent to some serial history.

3.4 Proving Full Invalidation Histories are View Serializable

In this section we show that if a history is accepted by a full invalidation TM automaton then that history is view serializable. We achieve this through seven lemmas and one corollary.

The most important of the lemmas are Lemma 4 and 8. Lemma 4 proves that all conflict serializable histories are also view serializable. Lemma 8 proves that if a full invalidation TM automaton accepts a history, then that history is conflict serializable. Putting these two results together, we have Theorem 9, which states that every history accepted by a full invalidation TM automaton is view serializable. The remainder of this section contains the proofs of these lemmas.

Lemma 1. *Two histories h and h' are view equivalent if and only if they involve the same set of transactions and $D(h) = D(h')$.*

Proof. The following proof starts by showing if h and h' are view equivalent, then $D(h) = D(h')$. It then shows that if $D(h) = D(h')$, then h and h' are view equivalent. Because the definition of view equivalence is nearly identical to the definition of D, the following proof is fairly straightforward.

- If h and h' are view equivalent, then $D(h) = D(h')$.

  - Suppose $(e_1, e_2) \in D(h)$. To show $D(h) = D(h')$ we must show that $(e_1, e_2) \in D(h')$. Because h and h' are view equivalent, any time a write to a variable that is subsequently followed by a read to the same variable must have the identical order in both histories. Because writes to variables followed by reads to the same variables are the only types of edges in D graphs, we know (e_1, e_2) is of this form. Therefore, if $(e_1, e_2) \in D(h)$ then it must also be in $D(h')$.

- If $D(h) = D(h')$, then h and h' are view equivalent. We proceed by case analysis on the three properties that must be maintained in order for two histories to be view equivalent.

  - Suppose transaction T_i in h reads an initial value (from T_0) for a variable x, then we must prove that transaction T_i also reads the same value in h'. T_0 for both h and h' will write all the values of the variables used in each history. Because $D(h) = D(h')$ the read after write edges from T_0 to T_i will be the same for both histories, therefore both histories will have the same read events from T_i for any variables that were written by T_0.

  - Suppose transaction T_i in h reads the value written by transaction T_j in h for a variable x, so does the transaction T_i in h'. Because $D(h) = D(h')$ reads after writes are identical for both histories. As such, any read event performed by T_i in history h from a value written by transaction T_j will be mirrored by transaction T_i in history h'.

  - Suppose transaction T_i in h is the final transaction to write the value for a variable x, so is the transaction T_i in h'. T_∞ for both h and h' will read all the values of the variables used in each history. Because $D(h) = D(h')$ the read after write edges for T_∞ will be the same for both histories, therefore both histories will have a commit event from T_i that contain the final write to x.

□

Lemma 2. *Two histories h and h' are conflict equivalent if and only if $D_2(h) = D_2(h')$.*

Proof. The following proof starts by showing if h and h' are conflict equivalent, then $D_2(h) = D_2(h')$. It then shows that if $D_2(h) = D_2(h')$, then h and h' are conflict equivalent.

- If h and h' are conflict equivalent, then $D_2(h) = D_2(h')$.
  - Suppose $(e_1, e_2) \in D_2(h)$. To show $D_2(h) = D_2(h')$ we must show that $(e_1, e_2) \in D_2(h')$. Because $(e_1, e_2) \in D_2(h)$, therefore $e_1 \prec\!\succ_h e_2$ and $e_1 <_h e_2$. Likewise, $e_1 \prec\!\succ_{h'} e_2$ and $e_1 <_{h'} e_2$ because h and h' are conflict equivalent. Hence, $(e_1, e_2) \in D_2(h')$.
- If $D_2(h) = D_2(h')$, then h and h' are conflict equivalent.
  - Suppose $e_1 \prec\!\succ_h e_2$. To show that h and h' are conflict equivalent, we must show that both h and h' have the same conflicts and these conflicts are ordered in the same way. In other words, $e_1 <_h e_2$ if and only if $e_1 <_{h'} e_2$. We proceed with case analysis on $e_1 \prec\!\succ_h e_2$, which yields two cases: (e_1, e_2) or (e_2, e_1).
    - Case 1: $(e_1, e_2) \in D_2(h)$. Because $(e_1, e_2) \in D_2(h)$, then $e_1 <_h e_2$. Because $D_2(h) = D_2(h')$, $(e_1, e_2) \in D_2(h')$. Because, $(e_1, e_2) \in D_2(h')$ then $e_1 <_{h'} e_2$
    - Case 2: $(e_2, e_1) \in D_2(h)$. Because $(e_2, e_1) \in D_2(h)$, then $e_2 <_h e_1$. Because $D_2(h) = D_2(h')$, $(e_2, e_1) \in D_2(h')$. Because, $(e_2, e_1) \in D_2(h')$ then $e_2 <_{h'} e_1$

□

Lemma 3. *If history h' is a permutation of history h, then $\forall e_1, e_2.\, e_1 \prec\!\succ_h e_2$ iff $e_1 \prec\!\succ_{h'} e_2$.*

Proof. This is a result of the symmetry of $\prec\!\succ$. Suppose $WW(e_1, e_2, h)$. Then we have two cases to consider: either $e_1 <_{h'} e_2$ or the reverse. In the former case, we have $WW(e_1, e_2, h')$ whereas in the later we have $WW(e_2, e_1, h')$. Either way, $e_1 \prec\!\succ_{h'} e_2$. Likewise, if $WR(e_1, e_2, h)$ then either $WR(e_1, e_2, h')$ or $WR(e_2, e_1, h')$. The same is true for RW conflicts. □

Lemma 4. *If a history h is conflict serializable, then h is view serializable.*

Proof. When h is conflict serializable, h is conflict equivalent to some serial history h', and, from Lemma 2, we have h and h' are conflict equivalent iff $D_2(h) = D_2(h')$. We need to show that when the preceding holds, h is view serializable. From Lemma 1, and following the same logic, if h is view serializable, then there is an h' that is serial and $D(h) = D(h')$. We proceed by showing that $D_2(h) = D_2(h')$ implies $D(h) = D(h')$.

To show $D(h) = D(h')$, we prove that all edges from one graph are present in the other. In other words, $(e_1, e_2) \in D(h)$ iff $(e_1, e_2) \in D(h')$.

- Suppose $(e_1, e_2) \in D(h)$. We now do case analysis on (e_1, e_2). However, because there is only one type of edge in $D(h)$, it must be an edge that denotes dependencies between different transactions.

  * Case $e_1 <_h e_2$, $e_1 = \langle T_1 commit, I, W\rangle$, $x \in dom(W)$, $e_2 = \langle T_2, x.read(v)\rangle$ (Dependencies between events in different transactions). We now perform case analysis on the possible transactions that can contain the events e_1 and e_2. There are three cases to analyze: (1) e_1 is not in T_0 and e_2 is not in T_∞, (2) e_1 is in T_0, and (3) e_2 is in T_∞.

    – Subcase $e_1 \notin T_0, e_2 \notin T_\infty$. Then $WR(e_1, e_2, h)$ and therefore $(e_1, e_2) \in D_2(h)$. Because $D_2(h) = D_2(h')$ and $e_1 \notin T_0, e_2 \notin T_\infty$, $(e_1, e_2) \in D(h')$.

    – Subcase $e_1 \in T_0$. Obviously, $e_1 <_{h'} e_2$. However, another commit event could occur before e_2, which we will call e_3, that would create the edge (e_3, e_2). If this edge existed in $D(h')$ it would also exist in $D(h)$. This is because h' is conflict equivalent to h and the edge (e_3, e_2) does come from transactions T_0 or T_∞, therefore the edge would exist in D_2 and therefore must exist in D (by conflict equivalence we have $D_2(h) = D_2(h')$ and $D(h) - \{T_0, T_\infty\} \subseteq D_2(h)$). However, this is a contradiction because both edges (e_1, e_2), $(e_3, e_2) \notin D(h)$ because a read event cannot read from multiple commit events. Therefore, e_2 must read from e_1, so $(e_1, e_2) \in D(h')$

  - Subcase $e_2 \in T_\infty$. This case follows from the same logic as the above case: the read event e_2 cannot read from multiple sources.

- Suppose $(e_1, e_2) \in D(h')$. We now do case analysis on (e_1, e_2). However, because there is only one type of edge in $D(h)$, it must be an edge that denotes dependencies between different transactions.

  * Case $e_1 <_{h'} e_2$, $e_1 = \langle T_1\, commit, I, W\rangle$, $x \in dom(W)$, $e_2 = \langle T_2, x.read(v)\rangle$ (Dependencies between events in different transactions). We now perform case analysis on the possible transactions that can contain the events e_1 and e_2. There are three cases to analyze: (1) e_1 is not in T_0 and e_2 is not in T_∞, (2) e_1 is in T_0, and (3) e_2 is in T_∞.

    - Subcase $e_1 \notin T_0, e_2 \notin T_\infty$. We have $WR(e_1, e_2, h')$, leading to $(e_1, e_2) \in D_2(h')$, therefore, $(e_1, e_2) \in D_2(h)$. Toward a contradiction, there could be a commit event, which we will call e_3, that could happen after e_1 but before e_2, creating an edge $(e_3, e_2) \in D_2(h')$. Because $(e_3, e_2) \in D_2(h')$, (e_3, e_2) is also $\in D_2(h)$. Because $(e_3, e_2) \in D_2(h)$, $(e_3, e_2) \in D(h)$. If $(e_3, e_2) \in D(h)$ then $(e_3, e_2) \in D(h')$, yet this is a contradiction because if $(e_3, e_2) \in D(h)$ then $(e_1, e_2) \notin D(h)$ because read-from events can only have a single source. However, from our assumption, $(e_1, e_2) \in D(h')$.
    - Subcase $e_1 \in T_0$. Obviously, $e_1 <_h e_2$. However, another commit event could occur before e_2, which we will call e_3, that would create the edge (e_3, e_2). If this edge existed in $D(h)$ it would also exist in $D(h')$. This is because h is conflict equivalent to h' and the edge (e_3, e_2) does come from transactions T_0 or T_∞, therefore the edge would exist in D_2 and therefore must exist in D (by conflict equivalence we have $D_2(h) = D_2(h')$ and $D(h') - \{T_0, T_\infty\} \subseteq D_2(h')$). However, this is a contradiction because both edges (e_1, e_2), $(e_3, e_2) \notin D(h')$ because a read event cannot read from multiple write events. Therefore, e_2 must read from e_1, so $(e_1, e_2) \in D(h)$

– Subcase $e_2 \in T_\infty$. This case follows from the same logic as the above case: the read event e_2 cannot read from multiple sources.

□

Lemma 5. *A history h is conflict serializable if and only if its lazy conflict graph, $G(h)$, is acyclic.*

Proof. The proof follows the general outline from [68] but is adapted to deal with a lazy conflict graph.

- Suppose that h is conflict serializable. Then there is a serial schedule, h', where the conflict events of h' have the same happens before ordering as h. It follows that $G(h) = G(h')$ by the definition of G. Furthermore, the lazy conflict graph $G(h')$ of the serial history h' is necessarily acyclic. Towards a contradiction, suppose $G(h')$ is cyclic. So there is a cycle $T_1, \ldots, T_n, T_1$ in $G(h')$. For each edge, (T_i, T_{i+1}), there is an event $e \in T_i$ that conflicts with an event $e' \in T_{i+1}$ and $e <_{h'} e'$. Because h' is serial, this requires that $T_i <_{h'} T_{i+1}$ (all the events in T_i happen before all the events in T_{i+1}). So we have $T_1 <_{h'} \cdots <_{h'} T_n <_{h'} T_1$. Then, because $<_{h'}$ is transitive, we have $T_1 <_{h'} T_n$. But we also have $T_n <_{h'} T_1$, which contradicts that $<_{h'}$ is a strict total order (in particular, the trichotomy law). Because $G(h')$ is acyclic and $G(h) = G(h')$, $G(h)$ is also acyclic.

- Suppose now that $G(h)$ is acyclic. Let us find a total order of the transactions which are compatible with the edges of $G(h)$. We do this by choosing one of the *topologically sorted* paths of $G(h)$.

  The resulting total order suggests a serial schedule h'. For h to be conflict equivalent with h', the following must hold: $\forall e_1, e_2.\ e_1, e_2 \in h$, if $e_1 \prec\!\succ_h e_2$ then $(e_1 <_h e_2)$ *iff* $(e_1 <_{h'} e_2)$. So, we must show that when $e_1 \prec\!\succ_h e_2$, that $(e_1 <_h e_2)$ *iff* $(e_1 <_{h'} e_2)$. We proceed by picking an arbitrary $e_1, e_2 \in h$ where $e_1 \prec\!\succ_h e_2$.

  * Suppose $e_1 <_h e_2$. Let T_1 such that $e_1 \in T_1$, T_2 such that $e_2 \in T_2$. Thus, $(T_1, T_2) \in$

$G(h)$. So, T_1 appears before T_2 in h', because h' is the transactional topological order of $G(h)$. Therefore, $e_1 <_{h'} e_2$.

* Suppose $e_1 <_{h'} e_2$. From $e_1 \prec\!\!\succ_h e_2$, we have:
  - $\mathit{conflicts{-}with}(e_1, e_2, h)$ which has $e_1 <_h e_2$ immediately.
  - $\mathit{conflicts{-}with}(e_2, e_1, h)$ which has $(T_2, T_1) \in G(h)$. Thus, $T_2 <_{h'} T_1$. So $e_2 <_{h'} e_1$ which is a contradiction.

It follows that h and h' are conflict equivalent.

□

Lemma 6. *Suppose* $\mathit{inval}(h)$ *and* $h = h_1 \cdot \langle T_1\ \mathit{commit}, I_1, W_1\rangle \cdot h_2$. *If* $\langle T_2, x.\mathit{read}(v_2)\rangle \in h_1$ *and* $x \in \mathit{dom}(W_1)$, *then* $\langle T_2\ \mathit{commit}, I_2, W_2\rangle \notin h_2$.

Proof. By the definition of the commit process in a full invalidation TM automaton in Figure 3.1: $h' = h \cdot \langle T\ \mathit{commit}, I, W\rangle$ only if $\forall T_x.T_x \in \mathit{active}(h) \wedge \mathit{valid}(T_x, h) \wedge (\mathit{writes}(T, h) \cap \mathit{reads}(T_x, h) \neq \emptyset) \rightarrow I = I \cup T_x$. Therefore, if T_2 committed, T_1 would be added to $T_2's$ invalidation set, thereby ensuring T_1 would not commit.

□

Corollary 7. *Commit events from both transactions* T_1 *and* T_2 *are not accepted by a full invalidation TM automaton if there is a RW conflict between* T_1 *and* T_2.

Proof. Immediate from Lemma 6. □

We now prove that if a full invalidation TM automaton accepts a history then that history is conflict serializable. Informally, we show this by proving that for all histories h, if h is accepted by a full invalidation TM automaton, then $G(h)$ is acyclic, therefore h is conflict serializable.

Lemma 8. *If* $\mathit{inval}(h)$, *then* $\mathit{conflict\_serial}(h)$.

Proof. By induction on the length of h.

- Basis Step ($length(h) = 0$): We have $conflict\_serial(h)$ because h's conflict graph, $G(h)$, has no vertices and no edges, so it is trivially acyclic.

- Inductive Step: From the inductive hypothesis, we assume that for any h, if $length(h) = k$ and $inval(h)$, then $conflict\_serial(h)$.

  Let h be a history where $length(h) = k+1$ and $inval(h)$. We need to show that $conflict\_serial(h)$. Because $length(h) = k + 1$, there exists h_1 and e_o such that $h = h_1 \cdot e_o$ and $length(h_1) = k$.

  We proceed by case analysis on e_o.

  * If e_o is an abort or read event then $G(h) = G(h_1)$ because $G(h)$ is only changed when a commit event is added to h (by the definition of a lazy conflict graph, Section 3.3).
  * Suppose e_o is a commit event, that is, it has the form $\langle T\ commit, I, W\rangle$. So $G(h)$ extends $G(h_1)$ with a new vertex labeled T and edges between T and vertices in $G(h_1)$ represent WW, WR, RW conflicts with other committed transactions, as depicted in Figure 3.2.

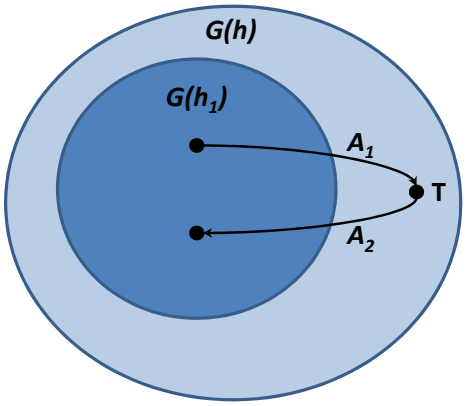


Figure 3.2: Creating a cycle in $G(h)$ from $G(h_1)$.

Because $G(h_1)$ is acyclic, $G(h)$ is also acyclic unless adding T to $G(h)$ creates at least two conflict edges in $G(h)$ such that one edge, named A_1, has T as its head and another edge, named A_2, has T as its tail. Formally, A_1 edges denote execution orders such that T must happen after some previously committed transaction in $G(h_1)$. A_2 edges

denote execution orders such that T must happen before some previously committed transaction in $G(h_1)$.

We proceed by case analysis on the conflict A_2.

- A_2 cannot be a WW conflict because WW conflicts create edges that order committed transactions to the actual execution order. Therefore, it is only possible for WW conflicts to create forward dependency edges (like A_1).
- A_2 cannot be a WR conflict for the same reason as a WW conflict.
- A_2 is a RW conflict. While a RW conflict could take the form of A_2, RW conflicts are rejected by the full invalidation TM automaton by Corollary 7. Therefore for A_2 to be a RW conflict is a contradiction.

□

Theorem 9. *If $inval(h)$, then h is view serializable.*

Proof. By Lemma 8, if $inval(h)$, then h is conflict serializable. By Lemma 4, if a history h is conflict serializable, then h is view serializable. Therefore, if $inval(h)$, then h is view serializable. □

This concludes the proof of our main theorem which proves that if a history is accepted by a full invalidation TM automaton, as defined in Section 3.2, then that history is both conflict and view serializable.

3.5 Future Work

While we believe proving that histories accepted by full invalidation TM automaton are both conflict and view serializable is a good first step to showing full invalidation is correct, we also believe several additional events should be modeled in order for our system to more closely match a real TM. We hope to extend the full invalidation model to include more complex events such as, the allocation and deallocation of memory elements, handling of user-level and system-level exceptions, and the arbitration of optimistic and pessimistic critical sections (i.e., lock-aware

transactional memory). Furthermore, a key trait of transactions is their atomic behavior when one transaction is nested within another transaction, known as transactional composition or nested transactions [41, 57, 61]. Future extensions of this work should include an analysis and extension of our automaton to support these types of transactions.

Chapter 4

Priority-Based Transactions

Parallel computers are the industry standard [5, 11, 2, 1] and, as such, programmers are writing more parallel programs [69, 48]. However, traditional parallel synchronization primitives such as locks, monitors, and semaphores, are exceptionally difficult to program correctly [46, 41, 4, 5]. These primitives exhibit nondeterministic behavior in their execution which give rise to a number of problems not found in serial executions like deadlocks, livelocks, lock convoys, and priority inversion [50, 41, 22]. Transactional memory shows promise in overcoming some conventional parallel programming problems and aims to simplify the task of writing correct, parallel code. However, even TM is susceptible to some of the problems that afflict conventional synchronization mechanisms.

Real-time systems or systems that have strict requirements for task execution, such as deadline-drive systems, usually guarantee such behavior through priority scheduling [56]. While substantive contention management (CM) research has been done, such as the work of Guerraoui et al. [36, 35] and Scherer and Scott [80, 79], their attention has been primarily focused on preventing starvation through fairness. In many systems preventing starvation may be sufficient, yet some systems, such as the ones described above (i.e., deadline-driven or real-time systems) require stronger guarantees. In these cases, user-defined priority-based transactions may be necessary.

In this chapter, we extend the prior TM contention management research of Guerraoui et al. and Scherer and Scott to support user-defined priority-based transactions inside of an invalidating STM, called TBoost.STM. TBoost.STM is a superset of InvalSTM (from Chapter 2) and as such

includes InvalSTM's full invalidation implementation as well as many other features not included in InvalSTM, such as the CM extension discussed in this chapter.

The research of Guerraoui et al. has noted that extending contention managers to include user-defined CM policies, such as user-defined priority-based transactions, is of importance [35]. We approach this by first presenting a brief background of TM aspects important to understanding the complexities of contention management. Next, we review and expand upon prior contention management work. We then show how conflict detection models play a significant role in the correctness and capability of priority-based transactional scheduling. We then build user-defined priority-based transactions demonstrating how contention management frameworks work with different conflict detection models. Last, we present our experimental results.

4.1 Contention Manager Background

Throughout this work we use the taxonomy presented by Harris, Larus, and Rajwar in their Transactional Memory texts [40, 54]. Some of these basic concepts (as well as some others) are briefly explained below.

4.1.1 Attacking and Victim Transactions

Following the terminology of Guerraoui et al. we refer to transactions which can identify a memory conflict in another active transaction as *attacking* transactions. Transactions which are targeted by attacking transactions are referred to as *victim* transactions [35].

An example of attacking and victim transactions is as follows. Consider transaction T_v, a victim transaction and transaction T_a, an attacking transaction. T_v writes to memory location L_0. T_a then tries to write to L_0. If our STM system only allows *single-writer* semantics, where only one active transaction can write to a given piece of memory, both active transactions T_a and T_v cannot concurrently write to L_0. Instead, once T_a attempts to write to L_0, the single-writer semantics require the W-W conflict at memory location L_0 between T_a and T_v be resolved. Handling this conflict makes T_a the attacking transaction and T_v the victim transaction. This is because T_a is

attempting to steal the memory location L_0 from transaction T_v, the transaction that already has optimistic ownership of L_0. T_v is the victim transaction because T_a may steal memory from T_v that T_v has previously, and exclusively, acquired.

4.1.2 Eager and Lazy Acquire

When transactions read and write to memory, they usually do so in different ways. For transaction T_1 to read memory location L_0, it can simply add some type of reference to itself (or at the memory location L_0) to indicate it is being read. Usually TM systems allow multiple readers of the same memory location. As such, any number of transactional readers may coexist for location L_0.

Written memory behaves in a different manner. Written memory must be exclusively acquired by a transaction at some point in the transaction's lifetime. Exclusive access to written memory is required because if multiple writers are allowed to concurrently update the same piece of memory there is a possibility that the resulting execution may be unserializable. To demonstrate this, consider transaction T_1 and T_2 both incrementing the integer stored at location L_0. L_0's initial value is 0. T_1 reads L_0 and increments it in a buffered local storage. Then T_2 reads L_0 and increments it, also in a buffered local storage. Both transactions read the value 0 and both temporarily store the value 1. When both transactions commit they both write the value 1 at location L_0. However, the correct result for both transactions incrementing location L_0 is 2. As such, exclusive memory access must be obtained by a transaction before it can commit writes. Two ways to acquire exclusive memory access is eagerly, also known as direct update, or lazily, also known as deferred update.

Eagerly acquired memory is usually exclusively obtained as soon as the transaction performs its write operation. Lazily acquired memory is usually exclusively obtained at commit-time. Each memory acquisition type has different benefits. Some early STM systems only allowed single-writer semantics and therefore had to perform eager memory acquisition. Other systems have begun allowing lazy memory acquisition as multiple-writers can result in different performance benefits. Some systems, such as FSTM, RSTM and DracoSTM allow for both eager and lazy acquires.

Eager acquires can improve system performance by allowing transactions to write their memory directly to its final location. If committed transactions are more common than aborted transactions this behavior can improve system performance because it is unnecessary to perform additional write operations during the transaction's commit phase. However, a disadvantage to eager write acquisition is that an attacking transaction may abort a conflicting victim transaction only to be aborted later by another transaction. This type of behavior, called *cascading aborts*, can limit transaction throughput and, in some rare cases, create livelock that will prevent forward progress altogether.

Lazy acquires can also improve performance by enabling serialized W-W conflicts as discussed in Chapter 3 and by overcoming the eager acquire scenario of cascading aborts that can cause livelocks. Lazy write acquisition can also stall writer transactions from committing so active reader transactions that have accessed the same memory as the writer transactions can commit before the writers, thereby increasing transaction throughput and reducing conflicts. However, lazy acquire can delay the notification of doomed transactions which can result in wasted work for all conflicting transactions.

4.1.3 Visible and Invisible Readers

When a transaction stores its read memory so other transactions can see it, it is called a *visible reader*. When a transaction stores its read memory so other transactions cannot see it, it is called an *invisible reader*. Visible and invisible readers are directly related to eager and lazy acquires. If the victim transactions for an eager or lazy write acquisition TM has visible readers, an attacking transaction can identify both W-W and W-R conflicts in the victim transactions at the time the attacking transaction acquires its write data. If the victim transactions have invisible readers, an attacking transaction can only identify W-W conflicts at write acquisition time. This postpones W-R and R-W conflict identification and resolution to some point later in the victim transactions' lifetime.

The principle difference in visible and invisible readers is in the conflict detection and resolu-

tion model. Visible readers allow attacking transactions to invalidate W-R conflicts, while invisible readers require the reading transactions to validate themselves.

4.1.4 Conflict Detection

The process of determining if a transaction can commit is called *conflict detection*. Many types of conflict detection exist. Two of the most common types of conflict detection are validation and invalidation.

When a TM system performs *validation*, each transaction's read and write set are checked for conflicts against global memory. If a conflict is found, the transaction (usually) must be aborted. This is because conflicts found using validation are those found with an active transaction and a previously committed transaction. In general, committed transactions cannot be uncommitted. As such, when a conflict is found, the active transaction must be aborted. Systems based entirely on validation usually employ invisible readers. Each transaction which reads an object does so in an invisible way, so other transactions cannot see these reads. When a transaction acquires a memory location for writing, the writer is unaware of other transactions that may be concurrently reading the same memory. As such, reader transactions must validate themselves at some point after writer transactions have exclusively acquired their written memory.

When a TM system performs *invalidation*, the TM checks one active transaction's write set (usually a transaction that is beginning its commit phase) for conflicts against all other active transactions. If a conflict is found the invalidating (attacking) transaction can either flag the active (victim) transaction as invalid, wait, or abort itself. Fully invalidating systems require that all transactions have visible readers, which allow the attacking transaction to identify and resolve W-W and W-R conflicts. Some partial invalidating systems employ invisible readers which allow attacking transactions to identify W-W conflicts but not W-R conflicts. In such partially invalidating systems, W-R conflicts are validated by readers at a later time.

Some systems perform both validation and invalidation, such as RSTM.v2 [58]. RSTM does this by allowing a limited number of visible readers to exist per memory location. If the

number of visible reader slots are filled, any additional readers become invisible. For example, when RSTM performs eager invalidation a transaction that writes to memory does so eagerly. Attacking transactions then invalidate any other existing write conflicts for the write memory as well as read conflicts through the memory location's visible reader list. If other transactions are invisibly reading the memory location, those invisible transactions must perform validation on themselves at a later time. RSTM performs lazy invalidation in a similar manner, except that memory is acquired in a lazy fashion, rather than an eager one. In both cases RSTM performs invalidation and validation based on the number of readers per memory location.

Validation and invalidation conflict detection are both practical for different classes of problems. When the number of active transactions are low and memory usage per transaction is high, invalidation may be preferred [24]. When the number of active transactions are high and memory usage per transaction is low, validation may be preferred. Neither type of conflict detection seems to perform universally better than the other. Though certain classes of problems perform better under different conflict detection models, the ramifications of these conflict detection models performing within strict user-defined priority-based transactional systems is unclear from prior research. The primary focus of this chapter is to clarify the advantages and disadvantages of validation and invalidation in user-defined priority-based TM environments.

4.2 User-Defined Priority-Based Transactions

Let us consider a case where user-defined priority-based transactions are needed. Figures 4.1 and 4.2 presents two conflicting transactions and how the varying conflict detection mechanisms, validation and invalidation, identify the same conflict. The examples show T_1 and T_2 working on the same integer, L_0. In both examples, T_1 modifies L_0 while T_2 reads it. T_1 commits before T_2 which causes T_2 to abort due to an inconsistent read.

Figure 4.1 shows how the conflicting transactions are handled using commit-time validation. When T_1 begins its commit phase, it checks the version of its local L_0 data against global memory. It sees the version number is the same and commits its changes to global memory (including

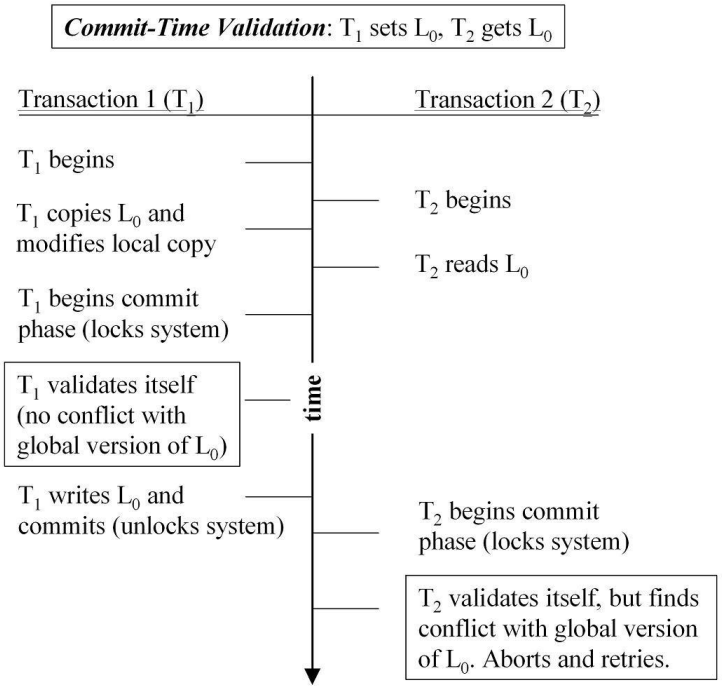


Figure 4.1: Transactions Conflicting in Commit-Time Validation.

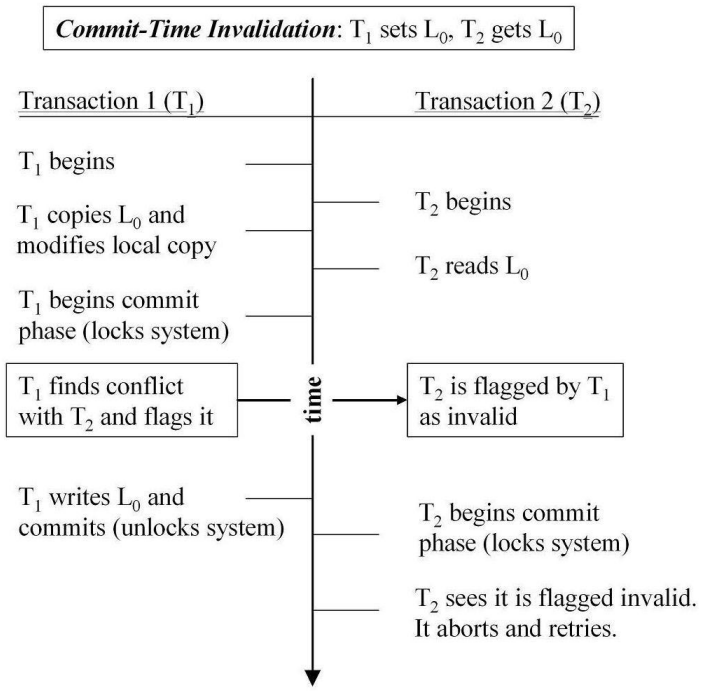


Figure 4.2: Transactions Conflicting in Commit-Time Invalidation.

incrementing the version). When T_2 begins its commit phase, it checks the version of its local L_0 data against global memory but sees its L_0 version is different than the global version. T_2 then aborts and restarts. Figure 4.2 shows how the conflicting transactions are handled using commit-time invalidation. When T_1 begins its commit phase it searches for conflicts with other active transactions. T_1 finds a conflict with T_2 and flags T_2 as invalid. T_1 then updates L_0 and commits. When T_2 performs its next transactional operation (its commit phase), it sees it has been flagged as invalid, aborts itself and restarts.

The commit-time behavior for both Figure 4.1's validation and Figure 4.2's invalidation is the CM behavior for TBoost.STM's iAggr explained in Chapter 2. [1] iAggr supports a first come, first serve policy such that the first transaction to reach the commit phase has precedence over other active transactions that the committing transaction conflicts with. TBoost.STM also supports priority-based contention management policies for transactions like Scherer and Scott's Karma or Polka policies, with priority based on read and write set size [80, 79] and Guerraoui et al.'s Greedy policy, with priority based on transaction timestamp [36, 35]. In addition, TBoost.STM also exposes a client-controlled transactional priority system for user-defined policies.

The transactions shown in Figures 4.1 and 4.2 cannot be managed by either DSTM's Karma or Polka policy, SXM's Greedy policy or TBoost.STM's iAggr policy such that T_2 would commit before T_1. Karma and Polka use read and write set size to determine priority and as T_1 and T_2 have the same size, it is assumed that since T_1 reaches the commit first, it would commit first. SXM's Greedy policy uses timestamp as priority and T_1's timestamp is earlier than T_2's, resulting in T_1's commit. Finally, TBoost.STM's iAggr, first come, first serve policy allows the first transaction of a set of conflicting transactions to commit. Because T_1 is the first of the two transactions to reach its commit phase it would be allowed to commit. As such, it is clear that if the user needed T_2 to commit before T_1, none of these CM policies would suffice. We now turn our attention to building a contention manager that enables T_2 to commit before T_1.

[1] Recall that TBoost.STM is a superset of InvalSTM. TBoostSTM implements all of InvalSTM's contention management policies and several other addition features not found in InvalSTM.

4.2.1 Adding User-Defined Priority to Transactions

Suppose there is a system-level requirement that says transaction T_2 must commit when transactions T_1 and T_2 are run concurrently. This type of system-level requirement is not unusual. A simple example showing the need for such a requirement is as follows.

Consider a customer connecting to a web server requesting information about a package she ordered. If we suppose that L_0 is the package history and T_1 is an update on its history and T_2 is a status of that history, it is now possible to see why a system designer would want T_2 to have priority over T_1. If T_2 is delayed by T_1, and T_1 is updated with high frequency due to substantial activity on the customer's history, T_2 may be delayed indefinitely. While T_2 is delayed, the customer is unable to see the status of her history.

Although this is a seemingly minor requirement, consider the effects if it was not in place and the web server handled business-to-business transactions. The customer checking status could be checking thousands or tens of thousands of orders. If the customer were requesting information about all of her orders, a minor delay in each request could result in an enormous delay for the entire history. As such, the system designer now has a practical reason to make T_2 always complete before T_1 when run together. Figure 4.3 demonstrates how this can be achieved in TBoost.STM.

Figure 4.3 shows the second transaction (T_2) raising its priority so it has a higher priority than the first transaction (T_1). While the transactions now have priority, TBoost.STM's contention manager must still be overloaded so it will acknowledge the user-defined priorities. TBoost.STM has two CM interfaces for handling conflict, `abort_before_commit()` which is used when TBoost.STM is performing commit-time validation and `permission_to_abort()` which is used when TBoost.STM is performing commit-time invalidation. Each must be appropriately overloaded depending on the conflict detection mechanism that is used.

Figure 4.4 extends TBoost.STM's CM for commit-time validation with user-defined priorities. The implementation of `abort_before_commit()` iterates through all active transactions, analyzing their priority against the committing transactions priority. If an active transaction is found with

```
native_trans<int> global_int;

//-----------------------------------------------------------------------------
// Transaction 1
//-----------------------------------------------------------------------------
void set_global(int val)
{
  atomic(t)
  {
    t.w(global_int).value() = val;
  } end_atom
}

//-----------------------------------------------------------------------------
// Transaction 2
//-----------------------------------------------------------------------------
void get_global(int val)
{
  atomic(t)
  {
    t.raise_priority(); // <- user priority
    t.w(global_int).value() = val;
  } end_atom
}
```

Figure 4.3: Get/Set Shared Integer with User-Defined Priority.

```
1 //--------------------------------------------------------------------------
2 // User-defined, derived CM abort_before_commit() for validation.
3 //--------------------------------------------------------------------------
4 class val_priority : public base_transaction
5 {
6 public:
7   //--------------------------------------------------------------------------
8   // abort_before_commit(), which is only used for validation, is called each
9   // time a transaction enters its commit phase. abort_before_commit() only
10  // checks the priority of transactions because in validating TMs, readers
11  // are invisible, so W-R memory conflicts between transactions cannot be
12  // identified by a committing transaction, they must be identified later
13  // when a transaction is validated at commit-time.
14  //--------------------------------------------------------------------------
15  virtual bool abort_before_commit(transaction const &t)
16  {
17    for (trans_container::const_iterator i = in_flight_trans().begin();
18         i != in_flight_trans().end(); ++i)
19    {
20      if (t.priority() < (*i)->priority()) return true;
21    }
22    return false;
23  }
24 };
25
26 //--------------------------------------------------------------------------
27 // Validation-based end_transaction() method
28 //--------------------------------------------------------------------------
29 state transaction::end_transaction()
30 {
31  if (cm_->abort_before_commit(*this))
32  {
33    lock_and_abort();
34    throw aborted_transaction_exception
35    ("aborting due to CM priority");
36  }
37  // ... rest of end transaction here
38 }
```

Figure 4.4: A Validating, Priority-Based, Transaction Scheduler.

a higher priority than the committing transaction, `abort_before_commit()` returns true which causes the committing transaction to abort. Otherwise, `abort_before_commit()` returns false allowing the committing transaction to proceed with its commit operation.

Figure 4.5 extends TBoost.STM's CM for commit-time invalidation with user-defined priorities and provides a brief overview of how the internal TBoost.STM implementation behaves with regard to TBoost.STM's invalidation interfaces. In Figure 4.5, the committing, attacking transaction first identifies a memory conflict. TBoost.STM's conflict handler then calls into the CM's `permission_to_abort()` API, passing both the attacking and victim transactions as parameters to the method. The user-defined `permission_to_abort()` then returns true if the attacking transaction has an equal or greater user-defined priority. In the case of the get and set example with transactions T_1 and T_2, the get transaction (T_2) would have a higher priority than the set transaction (T_1), therefore even if T_1 began its commit operation before T_2, T_1 would be forced to abort by the user-defined `permission_to_abort()` policy.

4.2.2 Summary

This section demonstrated how TBoost.STM's commit-time validation and invalidation contention management system could be overloaded to handle simple user-defined priority-based transactions. An important observation regarding commit-time validation is that it does not detect actual conflicts. Instead, because readers are invisible when using validation, TBoost.STM can only analyze priorities of transactions. As such, TBoost.STM's `abort_before_commit()` implementation as shown in Figure 4.4 does not abort a transaction based on a memory conflict. Instead, it aborts all committing transactions that have a lower priority than any in-flight transaction, even if no conflict exists between the transactions. This has the potential to severely limit transaction throughput for cases when transactional priority must be respected, but transactions have a minimal amount of contention between them.

On the other hand, when user-defined priority is employed using commit-time invalidation, only transactions that have actual memory conflicts with one another can result in aborts. If

```
1 //--------------------------------------------------------------------------
2 // User-defined, derived CM permission_to_abort() for invalidation.
3 //--------------------------------------------------------------------------
4 class inval_priority : public base_transaction
5 {
6 public:
7   //--------------------------------------------------------------------------
8   // permission_to_abort(), unlike abort_before_commit(), is only called when
9   // a conflict exists between lhs and rhs.
10  //--------------------------------------------------------------------------
11  virtual bool permission_to_abort(transaction const &lhs, transaction const &rhs)
12  {
13    return lhs.priority() >= rhs.priority();
14  }
15 };
16
17 //--------------------------------------------------------------------------
18 // invalidating write-write conflict method
19 //--------------------------------------------------------------------------
20 void transaction::abort_write_write_conflicts()
21 {
22   // iterate through all our written memory
23   for (iter i = w().begin(); w().end() != i; ++i)
24   {
25     // iterate through in flight transactions
26     for (iter j = in_flight_trans().begin(); j != in_flight_trans().end(); ++j)
27     {
28       // memory conflict?
29       if (j->w().end() != j->w().find(i))
30       {
31         if (cm_->permission_to_abort(*this, *j))
32         {
33           // add to flag_as_aborted list, flag
34           // only after all conflicts are found
35         }
36         else throw aborted_transaction_exception("abort due to CM priority");
37       }
38     }
39   }
40 }
```

Figure 4.5: An Invalidating, Priority-Based, Transaction Scheduler.

two or more transactions have differing priorities but access no common shared data, they will not be aborted in an invalidating TM (although they could be in a validating TM). This is why TBoost.STM's `permission_to_abort()` implementation as shown in Figure 4.5 only aborts transactions when actual memory conflicts exist between the transactions. As we will show in the upcoming sections, the differences in which validation and invalidation support user-defined priority-based transactions result in notably different transaction throughput.

4.3 A Real Example of User-Defined Priority-Based Transactions

In the following example, we present a problem where a specific transaction commit order is required for some concurrent transaction executions due to system-level constraints. In particular, priority-based transactions are used to ensure the correct system behavior is maintained during execution. Although the below example is applied to a specific domain, the same class of problem exists in a number of other areas such as medical, space, and defense systems. In each of these areas response time must be guaranteed within rigid parameters or certain severe consequences may result (e.g. irreversible patient damage or death, spacecraft failure and citizen or military loss of life). While the below example is given using a set of constraints that are applied to a single domain, this scenario might be thought of as a specific instance for an entire class of problems.

The following software system is implemented using three threads that access the same shared integer array with varying degrees of overlapping access between the threads. Below is a short description of each thread's behavior.

(1) Thread 1 (transaction T_1) iteratively reads each element of the integer array in a single transaction.

(2) Thread 2 (transaction T_2) executes one of several high-priority transactions based on the values read from thread 1. In addition, at a very infrequent rate, T_2 sometimes rereads the shared integer array.

(3) Thread 3 (transaction T_3) updates one integer element in the shared integer array as updates

occur.

The above scenario could be applied to a number of different practical examples. One such scenario is to consider the three threads working together as an automated stock market exchanger based on real-time trends. Each integer location represents a stock market value. Thread 1 and 3 perform relatively straight forward actions. Thread 1 makes a local copy of all the current stock values and passes them to thread 2. Thread 3 updates a specific stock in the shared array with the latest real-time data.

Thread 2's behavior, however, is more complex. Thread 2 uses the results sent to it by thread 1 to perform two different actions; (1) sell a stock or (2) buy a stock based on the comparison of the current values against historical data. Thread 2 always queries and processes the historical data before taking any action. The querying and comparison of historical data takes a certain amount of time. Generally, thread 2 accesses historical data before any datum within the current stock values has changed, so there are minimal negative side-effects from the execution of its polling process. However, in some special cases (e.g. selling or buying large amounts of stocks), thread's 2 pending action may have dire consequences if the preconditions of which its decision has been based off of have changed. Under such circumstances, the requirements of the software are such that thread 2 must reverify each of the current stock values to ensure the delay caused by analyzing historical data has not resulted in an unacceptable change in stock values. In these cases, thread 2 must reread all of the shared integer array values and validate them before performing its critical action.

In general, thread 2's transaction, T_2, has a higher priority than the other transactions. If T_2 is not executed quickly (i.e., if T_2 is not executed without being aborted), the overall system actions become slightly or extremely erroneous. This is because if the stock market prices change while T_2 is performing a stock trade action, the trade action may be reduced in the degree of profit it could have attained or, even worse, the trade action could result in a deficit even though the original data showed the trade action would result in some degree of profit.

Furthermore, if the shared integer array size is large, T_1's array read operation may be

starved by T_3's write operation. T_1 could be starved by T_3 because T_1's read operation performs substantially more work than T_3. Yet, T_3's write operation is likely to conflict with T_1's read, increasing the likelihood of a transactional conflict between T_1 and T_3, each time T_3 commits. If T_1 and T_3 are run concurrently in a repeating cycle, the previous examples of static priority shown in Figure 4.3 might cause T_1 or T_3 to starve, depending on which transaction received the lower priority. This is because T_3 would likely continually commit before T_1, resulting in either T_1 aborting if T_3 has a higher static priority or T_3 aborting if T_1 has a higher static priority. As such, a different priority-based solution must be implemented for this problem.

While either Scherer and Scott's Karma policy or Guerraoui et al.'s Greedy policy would prevent starvation between T_1 and T_3, neither would allow T_2 to take a natural higher priority than either T_1 and T_3. Since some cases exist where T_2 conflicts with T_3, some mechanism must be put in place to prevent T_3 from committing when it is conflicting with T_2. Therefore, to solve this problem elegantly we can use user-defined priority-based transactions to achieve the behavior that is required by the system-level requirements.

4.3.1 Dynamic Priority Assignment

A scheduling model where each task is prioritized based on how close it is to its deadline is called *dynamic priority scheduling* or *dynamic priority assignment* [56]. Dynamic priority assignment is preferred over the static priority assignment shown in Figure 4.3. We use a basic form of dynamic priority assignment for transactions T_1, T_2 and T_3 which increase their priority each time they are aborted. Our dynamic priority scheduling is similar to Ramanathan and Moncef's dynamic priority based scheduling (although considerably simplified) [75]. In Ramanathan and Moncef's dynamic priority algorithm, they increase priority as deadlines grow closer, we increase priority based on the number iterative aborts that a transaction suffers. Both algorithms have the same fundamental goal: as time moves forward and the task fails to complete, the priority of the task should increase. Ramanathan and Moncef's algorithm functions this way in order to meet time-critical deadlines, our algorithm does this to prevent starvation between transactions T_1 and

T_3.

The TBoost.STM code used for all three transactions is shown in Figure 4.6. Transaction T_1 copies the shared integer array into a local array. That local array is passed to transaction T_2. Transaction T_2 calls into an API that returns the appropriate action to take based on the current state of the stock prices compared to historical data. T_2 then checks to see if it needs to reverify the current data based on the time that has passed since the data was originally received and the action it is about to perform. If T_2 needs to reverify its data with the data currently stored in the shared data array, it can conflict with T_3 because T_3 may be updating the shared array. Transaction T_3 writes random locations in the shared integer array based on the changing stock values.

4.3.2 Priority-Based Transactions and False Positives

T_2, as shown in Figure 4.6, is set a priority based on the importance of the action it is about to perform. T_2 performs most of its reads and writes in isolation. Therefore, T_2 rarely has memory conflicts with other transactions. However, if commit-time validation is used when T_2 is executing the system will abort all committing transactions which have a lower priority than T_2, although most transactions will be free of memory conflicts with T_2. Although some conflicts may eventually occur (such as those from T_3), most of the aborts of T_1 and T_3 while T_2 is executing are unnecessary, because they do not actually conflict with T_2. In particular, T_1 never conflicts with T_2 because their conflicts are read-read conflicts which always result in serializable executions. These aborts can result in a significant performance penalty if the transactions are executed concurrently with high frequency, as shown in the upcoming experimental results section.

The invalidating system's priority scheduler shown in Figure 4.5 processes T_2 without aborting T_1 at all, and only aborts T_3 when it is necessary in order to preserve a serializable commit order. Recall that TBoost.STM's commit-time invalidation CM scheduler is only invoked when memory conflicts exist between two transactions. As such, even in the cases when T_2 must read the shared integer array but has yet to perform the reads, updates to the shared integer array by T_3 do not cause a conflict and therefore T_3 will be allowed to commit while T_2 is running concurrently as

```
native_trans<int> arr[100];

//------------------------------------------------------------------------------
// T1 copies all of the data from the shared integer array
//------------------------------------------------------------------------------
void get_arr(int out[])
{
  atomic(t)
  {
    for (int i = 0; i < 100; ++i) out[i] = t.read(arr[i]).value();
  }
  catch_before_retry{ t.raise_priority(); }
}

//------------------------------------------------------------------------------
// T2 reads the shared data handed to it by T1. It then proceeds by calling
// a method which determines what action to perform based on the current data
// and a collection of historical data. From this method, an action type is
// returned which is used to set the priority of the transaction. Next, T2
// determines if it needs to revalidate the previously read shared data against
// its current state. If so, T2 aborts if the values are different or proceeds
// if they are the same.
//------------------------------------------------------------------------------
int exe_task(int orig[])
{
  atomic(t)
  {
    int action = perform_action_using_historical_data(orig);
    t.set_priority( get_task_priority_from_action( action ) );

    if (need_to_revalidate_shared_data(action))
    {
      int out[100]; get_arr(out);
      // verify out and orig are the same, if not abort and retry
    }
    t.end_transaction();
    perform_action(action);
  }
  catch_before_retry { t.raise_priority(); }
}

//------------------------------------------------------------------------------
// T3 is called when the stock values change. The caller sends a value and an
// array location for the stock price and stock location in the shared array.
//------------------------------------------------------------------------------
void set_arr(int val, int loc)
{
  atomic(t)
  {
    t.w(arr[loc]).value() = val;
    t.end_transaction();
  }
  catch_before_retry { t.raise_priority(); }
}
```

Figure 4.6: Dynamic Priority-Assignment for Transactions T_1, T_2 and T_3.

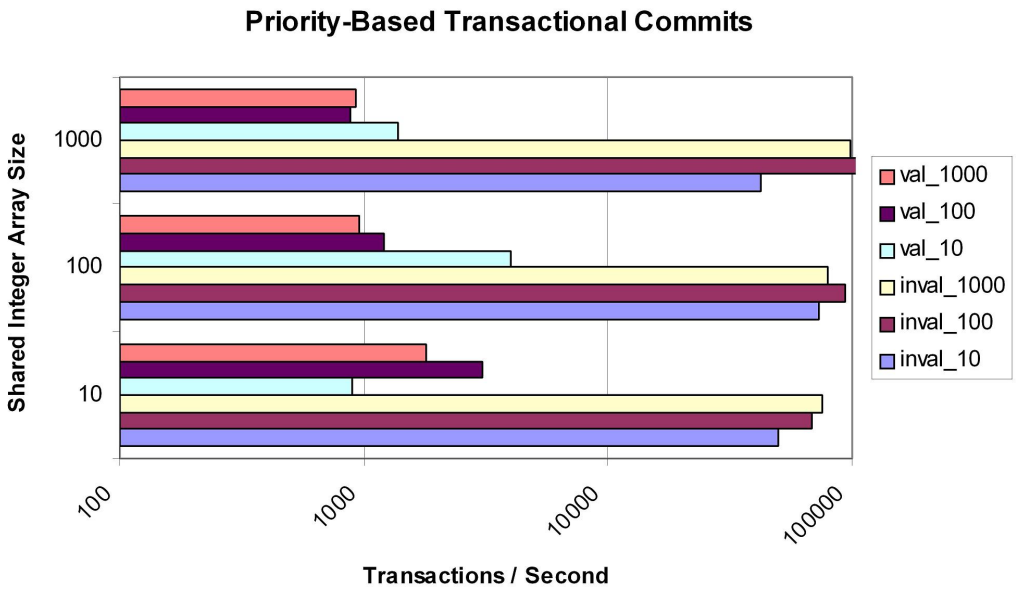
long as no conflicts exist between the two transactions. Once T_2 has already read the shared integer array, if T_3 tries to update a location and commit, T_3 will generally be forced to abort as to prevent it from causing T_2 to abort when it is about commit. From this context, the commit-time invalidation system can increase transaction throughput while still retaining a serializable execution. The result of this behavior of commit-time invalidation yields improved system performance when the transactions are executed concurrently with high frequency.

4.3.3 An Important Observation

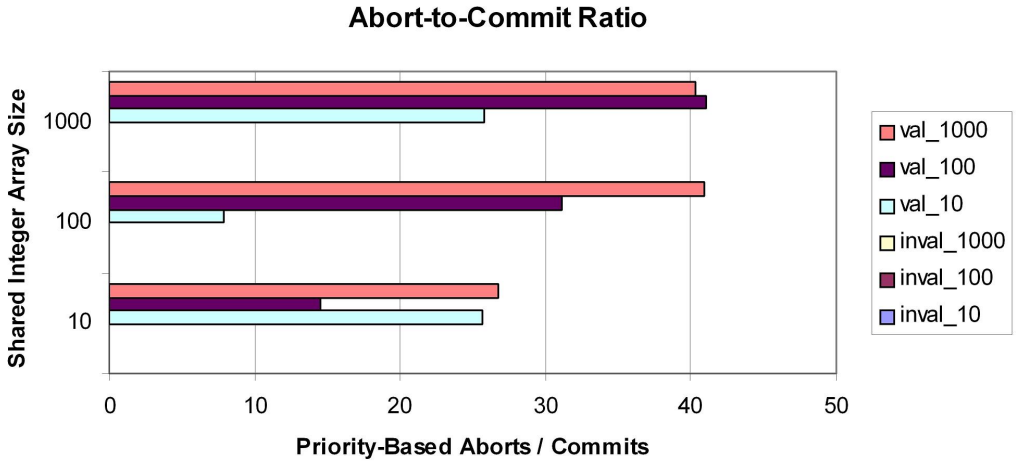
An important observation derived from the above scenario is that any system that does not perform full invalidation must abort all lower priority, committing transactions when higher priority transactions are active. Transactions that have a lower priority than another active transaction must be aborted in TMs that do not perform full invalidation because if they were not they might commit memory that could cause a conflict with another active, and higher priority, transaction. If this occurred, the higher priority transaction would be subsequently aborted causing priority inversion.

To demonstrate this, consider a system which performs eager invalidation with invisible readers. Eager invalidation with invisible readers requires that the readers validate themselves at commit-time. When invisible readers reach their commit phase, cases can exist when they must abort due to a previously committed write conflict. If such a case occurred, such that a high priority, invisible reader was aborted due to a previously committed lower priority writer, the system would exhibit priority inversion. To prevent this, any transaction that is not performing complete invalidation, must survey all in-flight transactions before committing. If a conflict is found between the committing transaction and another in-flight transaction with higher priority, the committing transaction must abort.

The effects of this observation seem to restrict the use of general purpose contention managers to systems outside of user-defined priority-based transactions. Prior contention management systems have been unrestrictive in that they have allowed portions of the conflict detection system to



(a) Priority-Based Transactional Commits



(b) Abort-to-Commit Ratio

Figure 4.7: Priority-Based Transactional Commits and Abort-to-Commit Ratio.

be performed by both validation and invalidation. The mixed validation and invalidation portions of fairness-oriented contention managers have been reasonable as the invalidated portions of the transaction help to identify manageable conflicts, while the validated portions of the transaction help scalability. Thus, contention managers that are driven toward fairness have previously allowed a mixture of validation and invalidation and it has been reasonable to do so.

Yet, user-defined priority-based transactions require more strict partial ordering commit behavior than fairness driven contention managers may be able to provide. A primary reason for this shortcoming is that validation suffers the above performance deficiencies for user-defined priority-based transactions due to the strict ordering requirement. As such, it is possible that a re-evaluation of contention management philosophy for systems that require user-defined priority-based transactions may be necessary. While a detailed analysis of this problem is outside the scope of this work, Spear et al. have performed some research in this vein, which heavily extends our original findings and contributes a number of new ideas [83].

4.4 Experimental Results

The experimental results presented in this section were based on running implementations derived from the example presented in Figure 4.6. All results were run on a 3.2 GHz 4-processor Intel Xeon with 16 GB of RAM. A brief summary of each thread's workload is listed below.

(1) Thread 1 (transaction T_1) reads each element of the integer array in a single transaction.

(2) Thread 2 (transaction T_2) simulates the execution of a high priority task by creating a high priority transaction, sleeping for a time and writing to memory different than T_1 and T_3. In some cases, T_2 is required to reread the shared integer array.

(3) Thread 3 (transaction T_3) updates one integer element in the shared integer array.

Program termination was controlled by a fixed number of successful iterations of T_1. When T_1 successfully completed N iterations, the program terminated. Any number of program termination

mechanisms could have been used, such as total duration, successful iterations of any thread's transaction, total number of aborts, etc. We arbitrarily chose to terminate the program based on T_1's successful commits.

Eighteen different execution configurations were tested. First, each execution had to successfully commit a specific number of T_1 transactions. We tested three variations of T_1 commits: 10^1 times, 10^2 times, and finally 10^3 times. For each execution, we tested both commit-time validation (labeled: val_10, val_100 and val_1000 in the graphs) and commit-time invalidation (labeled: inval_10, inval_100 and inval_1000 in the graphs). In addition, for each set of iterations, $T_1 = 10^1$, 10^2, or 10^3, and for both validation and invalidation, three different shared array sizes were used: 10^1, 10^2 and 10^3. The result is a nine-way performance analysis (three shared integer array sizes by three different T_1 successful completions) per conflict detection model, resulting in a total of eighteen execution configurations. We varied the array sizes in an attempt to see results that were both aligned and misaligned with the machine's caches.

The results shown in Figures 4.7(a) and 4.7(b) are as expected based on the prior analytical treatment of commit-time validation and commit-time invalidation. Commit-time validation performs roughly one to two orders of magnitude worse than commit-time invalidation because it must abort lower priority transactions when higher priority transactions are active even though in many cases these aborts may not be necessary. Furthermore, commit-time validation suffers roughly a 25-40 abort-to-commit ratio with two outliers of roughly 8 and 15 abort-to-commit ratios as shown in Figure 4.7(b). Commit-time invalidation consistently has less than 0.008 abort-to-commit ratio for all benchmarks, making the invalidation bars on the bar graph in Figure 4.7(b) so small that they cannot be seen. It is our opinion that the abort-to-commit ratio is the key performance statistic of this example. Commit-time validation performs poorly because of its relatively high abort-to-commit ratio when compared to commit-time invalidation.

4.5 Conclusion and Future Work

In this chapter we extended the prior work of Scherer and Scott and Guerraoui et al. by implementing user-defined contention managers. We found that the TM's conflict detection mechanisms can have a direct and notable affect on the overall performance of system when user-defined priority-based transactions are necessary. Ensuring strict partial ordering of competing tasks is a stronger requirement than fairness and, as such, some of the previous flexibility in contention management strategies cannot be applied to user-defined priority-based transactions without suffering notable performance degradations. As the requirements of user-defined priority-based transactions become more strict, they can cause splintering performance differences between conflict detection models. We reported on some of these performance differences in the experimental results section of this chapter. Commit-time invalidation was able to perform one to two orders of magnitude more work than commit-time validation.

We also noted what we believe is an important observation. We found that any system which supports user-defined priority-based transactions and does not employ full invalidation must abort any committing lower priority transactions if other higher priority transactions are in-flight regardless of whether these in-flight transactions present memory conflicts with the committing transaction. The degraded performance caused by aborting these transactions suggests a refocus of contention management philosophy for critical systems and might even suggest that validation not be used as the conflict detection mechanism for such systems.

We found that commit-time invalidation performs well for user-defined priority-based transactions, but it can incur substantial serialization overhead when a great deal of active transactions must be scanned for conflicts for the reasons explained in Chapter 2. As such, other speculative invalidation models should also be considered as alternative conflict detection mechanisms for critical systems. Also, we believe that whether or not the type of conflict detection should alternate at run-time should be investigated. While we have attempted to present a compelling argument of why user-defined priority-based transactions should be required, the examples we constructed were

fairly basic. A number of open questions remain regarding the analysis of real-time, critical, or deadline-driven system behavior in conjunction with user-defined priority-based transactions. An abundant body of research already exists in task scheduling and, as such, we encourage alternate solutions to be explored and compared against our initial findings.

Chapter 5

Lock-Aware Transactional Memory

A shortcoming of transactions is that they do not correctly interoperate with mutual exclusion locks without special effort. This interoperability failure is magnified by the prevalent use of locks in parallel software [51]. In this chapter, we argue, as others do, that for TM to become practical transactions and locks should be able to execute, concurrently and correctly, in the same program [3]. We call this type of TM, *lock-aware TM*, or LATM.

We are not the first to propose LATM. Other systems have been built which allow locks and transactions to run cooperatively together where their optimistic and pessimistic critical sections semantics are respected. Existing LATMs work by using run-time analysis to detect conflicts between locks and transactions to ensure their respective semantics are not violated [89, 91]. Unfortunately, with this type of run-time analysis, it may be impossible to guarantee that all of the potential conflicts that can exist between transactions and locks will be found before a conflict arises. Because of this, LATMs sometimes overestimate the actual conflicts that exist in an execution in order to preserve serializability, thereby limiting lock and transaction throughput to some subset of the system's full potential.

Our LATM uses full invalidation to manage conflicts between transactions and locks and programmer knowledge, in the form of programmer annotations, to indicate which transactions and locks may conflict as to reduce unnecessary serialization as found in earlier LATMs. The result is improved program performance. To the best of our knowledge, we are the first to propose using full invalidation for a LATM. However, other LATMs that typically use validation for transactional

conflict detection, use invalidation when employing conflict detection between locks and transactions or *irrevocable transactions*, transactions that cannot be aborted once they have begun [90].

The idea of using programmer knowledge to improve performance is not new to transactional memory; transactional boosting and open nesting are based on a similar principle [47, 62]. The programmer annotations of our LATM come in two forms: coarse-grained and fine-grained. Our coarse-grained LATM policy, called TM-lock protection, requires some programmer effort (e.g., one line of code per lock in the program) allowing the programmer to specify, at a high level (e.g., a C/C++ program's `main()` function) which locks may conflict with transactions. Our fine-grained LATM policy, called TX-lock protection, requires more programmer effort (e.g., one line of code per lock per transaction) but can theoretically yield higher concurrent transaction and lock throughput as only the critical sections that actually conflict are prevented from running concurrently.

The experimental results we present are surprising and extend our prior findings in [31]). Our coarse-grained policy is always faster than the prior state-of-the-art systems for our test cases, while our fine-grained policy is faster than prior work only in some cases. In other cases, our fine-grained policy performs worse than both prior work and our coarse-grained policy. These results are initially surprising, but we attempt to explain them in greater detail Section 5.5. We explain why the concurrent benefits of our fine-grained policy are not always observed and why our coarse-grained policy's concurrent throughput is more consistent for our test cases.

At the end of this chapter, we present a new area of research in contention management that we call *unified contention management* (UCM). Unified contention managers are an extension of prior CMs that handle the forward progress of both optimistic and pessimistic critical sections. Because prior work in contention management has been restricted to the forward progress of transactions, we believe UCM may cause subtle to substantial changes in the philosophy and strategies used in contention management.

As TMs become lock-aware, we suspect most TMs will use a UCM to provide some type of guarantee about the forward progress [66] of pessimistic (i.e., locks and, sometimes, transactions [43]) and optimistic (i.e., transactions and, sometimes, locks [72]) critical sections. In general,

contention management (CM) policies must arbitrate lock-based forward progress in a different manner than they arbitrate transaction-based forward progress because the underlying semantics of optimistic and pessimistic critical sections are different. As such, we briefly explore the importance and usefulness of UCMs with regard to efficient and practical LATMs. In this chapter, we make the following technical contributions:

(1) We present two novel LATM policies that are implemented using an extended form of full invalidation. With little programmer effort, our coarse-grained LATM policy (TM-lock protection) provides up to $\approx 1.4x$ performance improvement over prior state-of-the-art systems and is always faster than such systems for our experimental benchmarks. Our fine-grained LATM policy (TX-lock protection) requires more programmer effort, but provides up to $\approx 2x$ performance improvements over prior research for select experiments.

(2) We provide and discuss surprising experimental results of our LATM policies. In particular, we show that our coarse-grained LATM policy is usually faster than our fine-grained LATM policy. These results are surprising because our fine-grained LATM policy can theoretically increase concurrency beyond what is possible for our coarse-grained LATM policy.

(3) We conclude with the presentation of a new area of research in contention management, what we call unified contention management (UCM). For LATMs, UCMs provide the mechanisms necessary to ensure forward progress of both pessimistic and optimistic critical sections, alike.

5.1 Background

When transactions and locks are executed concurrently in TMs where transactions are not made aware of locks, what we call *non-lock-aware transactional memories* or *non-LATMs*, program execution can behave erratically due to the differences in the critical section semantics of locks and transactions [76, 89, 91]. Mutual exclusion locks generally use pessimistic critical sections that are limited to one thread of execution [20, 92]. Transactions generally use optimistic critical

sections that support unlimited concurrent execution and resolve serializability issues in the conflict detection stage of the TM [60, 71]. Optimistic and pessimistic critical sections conflict because their semantics differ; an example of this is demonstrated in Figure 5.1.

In non-LATMs, when threads 1 and 2 are executed concurrently and in the sequence shown in Figure 5.1 their swap behaviors can produce in an incorrect result. Both threads 1 and 2 implement a swap function. Thread 1 uses a lock, while thread 2 uses a transaction. Both of the swap implementations are correct when run in isolation, however, when they are run together their concurrent execution can be erroneous.

Consider a weakly isolated TM using direct update where the initial state of the program is `x = 1` and `y = 2`. When correctly swapped, `x = 2` and `y = 1`. Thread 1 starts by setting `tmp1 = 1` and `x = 2`. Thread 2 then sets `tmp2 = x` where `x = 2` and `x = y` where `y = 2`. Thread 1 then sets `y = tmp1` where `tmp1 = 1`. Thread 2 sets `y = tmp2` where `tmp2 = 2`. The resulting state (`x = 2, y = 2`) is incorrect and no possible number of consecutive swaps could have caused such a result. Furthermore, the results are the same if the TM used deferred update.

5.1.1 Classifying Transaction-Lock Failures

Volos et al. have identified five pathological ways that transactions and locks can interact, each of which produce a specific type of error. The pathological behaviors identified by Volos et al. are: blocking, livelock, deadlock, early release, and invisible locking [89].

- Blocking can occur when a lock inside a transaction (LiT) is not immediately acquired. In certain TMs, transactions must terminate if the thread in which they are running is context-switched out by the OS. In such systems, LiTs that do not obtain a lock before being context-switched out are aborted. In these TMs, forward progress may be stalled for as long as the context-switched aborts are repeated.

- Livelocks can occur when threads try to acquire locks outside of transactions (LoT) via spinning. In some TMs, spinning to acquire locks [7] prevents locks that have been obtained

```
1   //--------------------------
2   // Starting: x = 1, y = 2
3   //--------------------------
4
5   //--------------------------
6   // Thread 1
7   //--------------------------
8   lock(L);
9   int tmp1 = x;   // tmp1 = 1
10  x = y;          // x = 2
11                                  //--------------------------
12                                  // Thread 2
13                                  //--------------------------
14                                  atomic
15                                  {
16                                     int tmp2 = x;    // tmp2 = 2
17                                     x = y;           // x = 2
18
19  y = tmp1;       // y = 1
20  unlock(L);
21
22                                     y = tmp2;        // y = 2
23                                  }
24
25  //--------------------------
26  // Ending: x = 2, y = 2
27  //--------------------------
```

Figure 5.1: Lock and Transaction Swap Violation.

within a transaction from being released. This behavior can cause livelock situations to arise when a transaction tries to commit and fails to release the locks it has acquired.

- Deadlocks occur in a variety of ways through the interaction of transactions and locks. One example is when a transaction is aborted after a lock within it has been acquired, but before it is released. It seems that all non-LATMs are susceptible to these types of deadlock.

- Early release can occur when a transaction releases a lock that was obtained before the transaction was started. If such a transaction is subsequently retried, the lock is released multiple times resulting in a potentially inconsistent program state. Early release behaviors can also lead to deadlocks.

- Invisible locking occurs in lazy acquire (or deferred update) systems when locks inside of transactions are obtained at commit-time, rather than when the lock operations are executed. This behavior can cause locks to become optimistic, altering their (potentially necessary) pessimistic semantics.

5.1.2 Preventing Transaction-Lock Violations

When locks and transactions are executed in non-LATMs the program may be executed in such a way that the resulting state is not serializable. The reason for this is fairly straightforward: locks and transactions can be implemented such that their critical section semantics are fundamentally different. In general, locks use pessimistic critical sections, such that only one thread can execute a critical section protected by a mutual exclusion lock at a time [20]. Transactions, on the other hand, generally use optimistic critical sections that can allow a potentially unlimited number of threads to execute them simultaneously. If problems arise, the TM unwinds some number of transactions to ensure the total commit order of the transactions is serializable.

One way to prevent violations between the concurrent execution of transactions and locks is to make transactions aware of locks, such that when a lock's critical section is executed transactions that might access the same shared data, are postponed. In short, if the property of mutual exclusion

is not violated by transactions – execution of pessimistic critical sections are limited to a single thread of execution [20] – transactions and locks will behave in a non-pathological manner when run concurrently.

Our system avoids the five pathological behaviors found by Volos et al. in the following way. We prevent deadlock and invisible locking by ensuring transactions do not violate the mutual exclusion property of locks using our granularized LATM policies and local knowledge of these locks (details to follow). Blocking and livelocks are avoided by the STM's implementation which we extended as explained in their prior work [26]. The final pathological behavior, early release, is deemed illegal and caught at run-time. Further details are provided in Section 5.4.

5.1.3 No Semantics for Data Races

Our LATM system gives no semantics for data races. A particular type of optimization that can be found in highly refined lock-based concurrent software is the intentional dismissal of locks around shared data. These cases generally exist because the programmer believes certain shared data can be accessed inconsistently without causing a run-time error (i.e., reading a stale value does not cause an inconsistent program state). A simple example of this type of optimization is below.

```
1  Obj* ptr = NULL;
2  for (; ptr == NULL; ptr = sharedPtr) {}
3  ptr->foo();
```

Here the programmer spins until `sharedPtr != NULL`. The primary assumption is that the program will never deallocate nor change `sharedPtr` after it has been constructed. However, this assumption is based on the current state of the program. As the program evolves, these assumptions may no longer hold and inconsistencies may arise that the programmer had not foreseen. For example, if the original `sharedPtr` construction code was changed to the following, the above program could cause an invalid memory access.

```
1   lock(protectSharedPtr);
2   sharedPtr = new Obj();
3
4   if (conditionA) {
5      delete sharedPtr;
6      sharedPtr = new Obj(fromA);
7   }
8
9   unlock(protectSharedPtr);
```

Data races violate the fundamental property of mutual exclusion. We, along with Boehm and Adve [10], advocate for race-free concurrent programs. For our LATM system, all data races are considered harmful and we therefore give no LATM semantics to their execution.

5.2 Related Work

This section briefly presents prior state-of-the-art LATM research [89, 91]. In particular, we provide an overview of Volos et al. and Ziarek et al.'s research in contrast with our own.

The systems of Volos et al. and Ziarek et al. identify and resolve conflicts between locks and transactions entirely at run-time. While run-time identification of conflicts between locks and transactions has the advantage of not requiring the programmer to write additional code for such purposes, it has the disadvantage of being overly conservative which can negatively impact performance.

Our system, on the other hand, requires that the programmer annotate his or her software to identify potential conflicts that exist between transactions and locks. The system then uses these programmer annotations to determine which transactions and locks cannot run concurrently and prevents such behavior at run-time. As we demonstrate in Section 5.4, run-time discovery of conflicts between locks and transactions is insufficient to avoid deadlocks. Because of this, systems that only use run-time conflict discovery between transactions and locks must be overly conservative

to ensure program correctness or, if they are not overly conservative, are susceptible to deadlock. Because our system uses programmer annotations to identify all of the conflicts that exist between transactions and locks at compile-time, the system can use precise definitions of conflicts to limit only those critical sections that actually conflict with one another, thereby maximizing concurrent throughput as much as possible.

5.2.1 TxLocks

The work of Volos et al. identified the five transaction-lock pathologies presented in Section 5.1. To overcome these pathologies, Volos et al. modified OpenSolaris and implemented what they call *TxLocks*. Below we describe how TxLocks handles each pathology. To overcome blocking, TxLocks allows transactions to be suspended without termination which prevents blocking from occurring and is trivial to prove correct. TxLocks uses *deferred unlock* to prevent the early release pathology. Deferred unlock allows a transaction to obtain and hold a lock until the transaction commits, even if the lock API has called for its release. While this behavior prevents lock-based critical sections from being violated within a transaction (e.g., early release) it does not prevent deadlocks (details to follow). TxLocks avoids invisible locking by using *escape actions* that allow LiT critical sections to remain pessimistic while inside of transactions. Escape actions allow lock-based operations within transactions to be seen immediately rather than at commit-time which avoids the invisible locking pathology. Escape actions also avoid livelock and some deadlock behaviors. When TxLocks encounters a deadlock it uses a conflict resolution mechanism that breaks the deadlock by stealing locks from threads.

A key difference between TxLocks and our system is that TxLocks does not disallow deadlocks as our approach does. Instead, TxLocks breaks deadlocks when they occur at run-time. Unfortunately, breaking deadlocks can have negative side-effects. Deadlocks occur when two or more processes hold a resource the other requires to make forward progress. To break deadlocks, resources are stolen from a process and given to another. However, critical section operations may have already been partially executed by a process that is stolen from and stalled. Once a thread's re-

sources have been stolen, the partial effects of its operations may be seen by other threads resulting in inconsistent program states.

5.2.2 P-SLE and Atomic Serialization

Ziarek et al.'s LATM is implemented in Java and introduces two concepts: *pure-software lock elision* (P-SLE) and *atomic serialization*. With the exception of the early release behavior, all of the previous pathologies described earlier are avoided with P-SLE because P-SLE converts locks into transactions. Because no locks exist in P-SLE, the pathologies cannot occur. Furthermore, the early release behavior is not possible in Ziarek et al.'s implementation because its signature is illegal in Java.

However, as noted by the authors, locks cannot always be converted into optimistic transactions. One such example is when a lock-based critical section performs I/O which must be run without transactional interference. P-SLE handles this by reinstating all locks and using a single global lock to serialize transaction execution. This behavior, called *atomic serialization*, guarantees the mutual exclusion property is maintained by requiring that each transaction obtain a shared global lock to execute [91]. While atomic serialization ensures a program execution will be serializable, it can result in program inefficiencies because it requires that all transactions execute serially. This restriction may be overly conservative in certain executions and therefore degrade performance in such cases.

5.3 Locks Outside of Transactions (LoT)

In this section, we discuss locks outside of transactions or LoTs. LoTs are scenarios where a lock-based critical section is executed in one thread while a transaction-based critical section is concurrently executed in another thread. LoTs require special handling to ensure concurrently executed locks and transactions do not access the same shared memory, since such behavior could result in inconsistent execution as demonstrated in Section 5.1.

Figure 5.3 presents a LoT example used throughout this section. Six threads are used in the

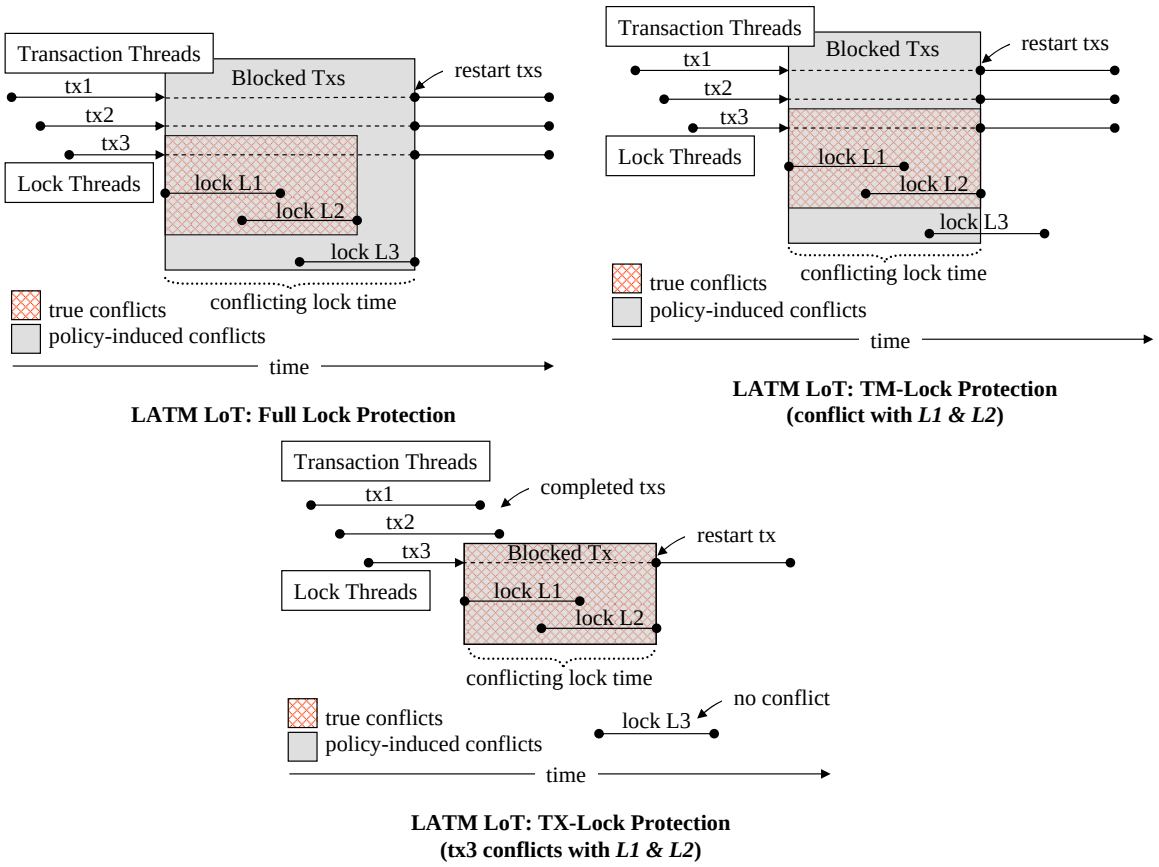


Figure 5.2: LoT Full (TxLocks), TM-, and TX-Lock Protection.

example that simultaneously execute six different functions. Three of the functions use transactions: thread T_1 runs `tx1()`, thread T_2 runs `tx2()`, and thread T_3 runs `tx3()`. Three of the functions use locks: thread T_4 runs `lock1()`, thread T_5 runs `lock2()`, and thread T_6 runs `lock3()`. The example uses a function called `no_conflict()` when a critical section executes code that uses shared data that does not conflict with any of the other critical sections.

In the example, thread T_3's function `tx3()` conflicts with both thread T_4 and thread T_5's locking functions (`lock1()` and `lock2()`). Threads T_1, T_2 and T_6 do not exhibit any conflicts, thus their calls to `no_conflict()`, but are necessary to demonstrate how the different LoT policies behave.

5.3.1 LoT Full Lock Protection and TxLocks

Our largest grained policy for transaction and lock cooperation, called full lock protection, forces all transactions to commit or abort before a lock's critical section is executed. LoT full lock protection is equivalent to Volos et al.'s TxLocks [89] and requires no programmer annotations because it is assumed all lock-based critical sections conflict with transaction-based critical sections. Locks are protected from violations from transactions because transactions are not allowed to run alongside them. The system stalls transactions until all lock-based critical sections are complete.

LoT full lock protection is overly conservative. Because no information is provided regarding potentially conflicting shared memory accesses within transactions, each time a lock-based critical section is executed transactions must be stalled. The maximum concurrent lock and transaction throughput achievable by LoT full lock protection (and TxLocks) at any given point in time is:

$$m(LoT_{fl}) = L_n \oplus T$$

L_n is the maximum number of lock-based critical sections that do not conflict with one another and T is the maximum number of transactions that can be executed, which do not conflict with one another. Because of the way LoT full lock protection behaves, only one type of critical section can be executed, locks or transactions, but not both. The symbol $\oplus$ in our equation is defined

```
1  //---------------------------
2  // Thread T1
3  //---------------------------
4  void tx1() { atomic(t) { no_conflict(); } }
5
6  //---------------------------
7  // Thread T2
8  //---------------------------
9  void tx2() { atomic(t) { no_conflict(); } }
10
11 //---------------------------
12 // Thread T3
13 //---------------------------
14 void tx3() {
15    atomic(t) {
16       for (int i=0; i < N; ++i) {
17          ++t.w(arr1[i]).value();
18          ++t.w(arr2[i]).value();
19       }
20    } end_atom
21 }
22
23 //---------------------------
24 // Thread T4
25 //---------------------------
26 int lock1() {
27    lock(L1); int sum = 0;
28    for (int i=0; i < N; ++i) sum += arr1[i];
29    unlock(L1); return sum;
30 }
31
32 //---------------------------
33 // Thread T5
34 //---------------------------
35 int lock2() {
36    lock(L2); int sum = 0;
37    for (int i=0; i < N; ++i) sum += arr2[i];
38    unlock(L2); return sum;
39 }
40
41 //---------------------------
42 // Thread T6
43 //---------------------------
44 int lock3() { lock(L3); no_conflict(); unlock(L3); }
```

Figure 5.3: Six Threaded LoT Example.

similarly to XOR. We define $\oplus$ to mean either the right side or the left side of the expression is used, but not both. Figure 5.2 presents a visualization of LoT full lock protection under the six threaded model. $T_1 - T_3$ are blocked for the duration of the critical sections of $T_4 - T_6$, even though T_6's critical section does not interfere with any of the transactions.

5.3.2 LoT TM-Lock Protection

TM-lock protection, our medium grained policy, requires some program annotations of the conflicts between transactions and locks, but can result in increased concurrent throughput over full lock protection when non-conflicting transactions and locks are concurrently executed. TM-lock protection works in the following way. The programmer identifies and specifies which locks guard shared data that is also accessed by at least one transaction. These programmer identified conflicts inform the TM system that if critical sections of the specified locks are executed concurrently with a transaction, an inconsistent program state could arise. Therefore, when a conflicting lock is acquired at run-time, the TM system aborts or commits all in-flight transactions and then prevents any new transactions from starting until after the conflicting lock-based critical section has completed its execution. Locks that do not conflict with any transaction do not cause stalls, which is an improvement over full lock protection. TM-lock protection's maximum critical section throughput can be expressed as follows:

$$m(LoT_{tm}) = L_{nl} \oplus (L_{na} + T)$$

L_{nl} is the total number of locks that do not conflict with one another, but do conflict with transactions. L_{na} is the total number of locks that do not conflict with one another and do not conflict with transactions. T is the maximum number of transactions that can be executed, which do not conflict with one another. In the six threaded example, TM-lock protection avoids unnecessary transaction stalling when T_6 is executing. As can be seen in Figure 5.2, TM-lock protection shortens the overall TM run-time compared to full lock protection by allowing $T_1 - T_3$ to restart their transactions as soon as `L2`'s critical section is completed.

5.3.3 LoT TX-Lock Protection

TX-lock protection, our smallest grained policy, requires local knowledge of locking conflicts as they exist per transaction. However, TX-lock protection has the highest potential concurrent throughput of the three policies. By precisely identifying which locks conflict with transactions, the system only stalls transactions when a lock-based critical section is executing that the programmer has specifically declared as a conflict as shown in Figure 5.2. LoT TX-lock protection increases concurrency potential when compared to LoT TM-lock protection, because it only stalls transaction and lock execution when a one-to-one conflict exists between the transaction and the lock. The following expression represents the maximum concurrent execution of locks and transactions at any given point in time for LoT TX-lock protection:

$$m(LoT_{tx}) = C_{lt} + L_{na} + T_{na}$$

C_{lt} is the largest system selected set of locks and transactions that can be run concurrently without containing any overlapping conflicting critical sections. L_{na} is the total number of locks that do not conflict with one another and have not been flagged as conflicting with any transaction. T_{na} are the transactions which do not conflict with any locks or other transactions. In the six threaded example, TX-lock protection stalls thread T_3 when the critical sections of `L1` and `L2` are executing. In this example, TX-lock protection's policy-induced conflict time is equivalent to the actual conflict time between critical sections. In other words, once a conflicting lock-based critical section has completed, the conflicting transactions are free to continue execution, and any other transactions which were non-conflicting were not impeded by the lock-based critical section's execution at all.

5.4 Locks Inside of Transactions (LiT)

In this section, we discuss locks inside of transactions or LiTs. LiTs are scenarios where a lock's pessimistic critical section is executed partially or completely inside a transaction. Our system only supports two of three possible LiT scenarios: (1) a lock-based critical section is placed

```
1   //---------------------------       //---------------------------
2   // Thread T1                        // Thread T2
3   //---------------------------       //---------------------------
4   lock(L1);                           atomic(Tx2) {
5
6                                          // Tx2 becomes irrevocable
7
8                                          lock(L1);
9   atomic(Tx1) {
10
11     ...                                 ...
12
13     unlock(L1);                         ...
14
15     ...                                 unlock(L1);
16  }                                   }
```

Figure 5.4: Early Release Deadlock.

entirely inside a transaction-based critical section or (2) a lock-based critical section begins inside a transaction-based critical section and ends after the transaction-based critical section ends. The third scenario, where a lock-based critical section begins before a transaction-based critical section starts, and the lock-based critical section ends inside the transaction-based critical section (known as early release [89]) is disallowed because it can cause deadlocks. This is a fundamental departure of our design from prior work. P-SLE does not deal with early release because it is illegal in Java. TxLocks allows it and attempts to break the deadlocks it creates. We believe that breaking deadlocks may have side-effects as we explained in Section 5.2.1.

5.4.1 Early Release Deadlocks in LiTs

To demonstrate how early release can cause deadlocks, consider Figure 5.4. Thread T_1 obtains lock L_1 while thread T_2 concurrently starts transaction Tx_2. Transaction Tx_2 is immediately made irrevocable, which means Tx_2 cannot be aborted (details to follow). Tx_2 then tries and fails to acquire lock L_1 because thread T_1 has already acquired it. Transaction Tx_2 then stalls, waiting for lock L_1.

Because lock L_1 is released inside of Tx_1, Tx_1 must be made irrevocable before L_1 is released.

If this were not done, the portion of L_1's pessimistic critical section inside of transaction Tx_1 could be retried. If L_1 critical section were retried it would violate the mutual exclusion property of the pessimistic critical section. However, because Tx_2 is already an irrevocable transaction, and at most only one irrevocable transaction can execute, Tx_1 cannot be made irrevocable. [1] Therefore, Tx_1 must stall until irrevocable transaction Tx_2 completes. However, Tx_2 is already stalled until lock L_1 is released, which will not happen until transaction Tx_1 completes. In summary, Tx_1 cannot make forward progress because it depends in Tx_2 and Tx_2 cannot make forward progress because it depends on Tx_1. The system is now deadlocked.

5.4.2 Irrevocable and Isolated Transactions

When lock-based critical sections are placed inside of transactions they should behave like normal locks. In other words, the mutual exclusion property of locks should be maintained when such locks are placed inside of transactions to ensure locks behave as they would normally. To support this property and because locks do not have failure atomicity (i.e., they do not support the property of side-effect free failures such as those found in transactions [8]), the transactions containing locks must be promoted to, at least, irrevocable status.

Irrevocable transactions are transactions that cannot be aborted. Irrevocable transactions (also known as inevitable transactions) are thoroughly investigated by Welc et al. and Spear et al. [90, 85] and are useful in a number of interesting scenarios. We extend the practical use of irrevocable transactions to that of ensuring lock-based pessimistic critical sections maintain their mutual exclusion property even when placed within transactions. In addition, we use irrevocable transactions to help create *composable locks* within transactions. Composable locks are locks that are acquired together to produce some meaningful execution that they would not be able to produce if acquired in isolation. Composed locks are described in more detail in the following section.

We also introduce a new type of transaction, which we call an *isolated transaction*. Isolated

[1] Two irrevocable transactions cannot execute concurrently because they are not guaranteed to be free of conflicts between each other [90, 85].

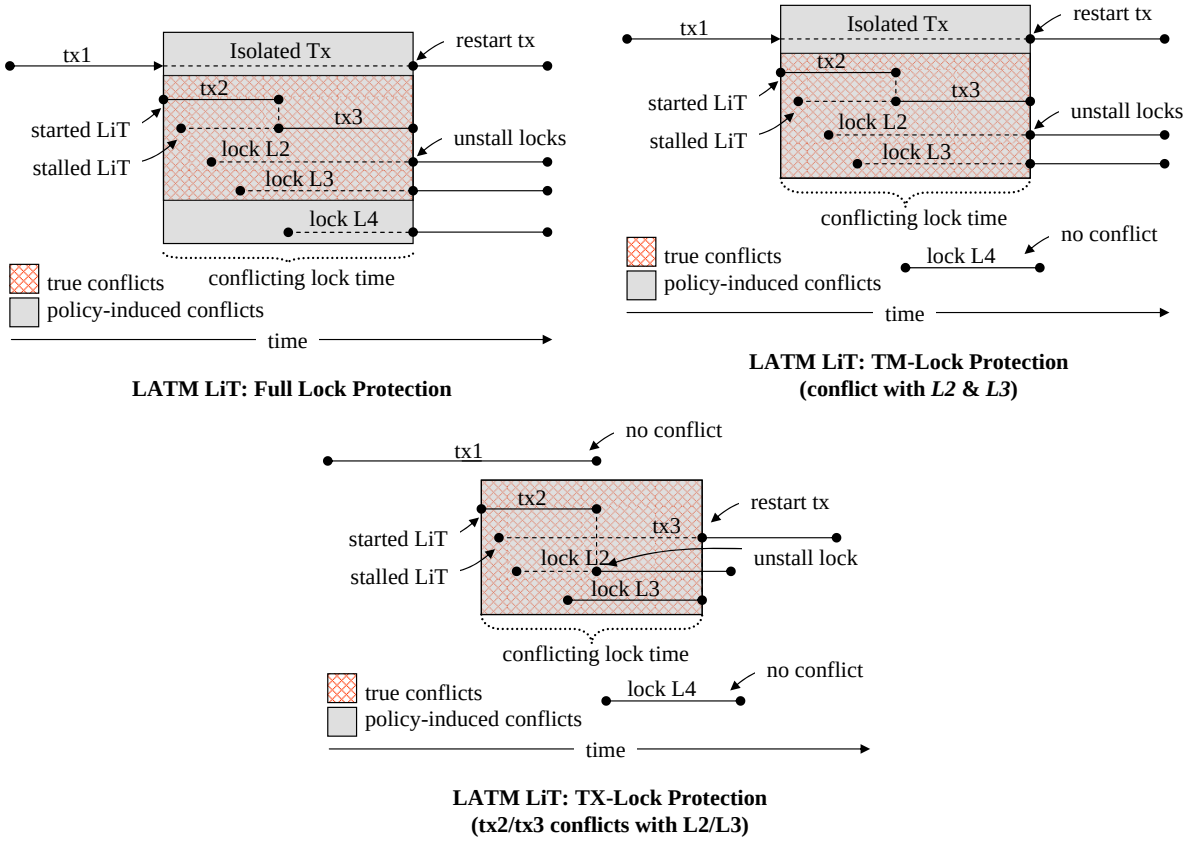


Figure 5.5: LiT Full (Atomic Serialization), TM-, and TX-Lock Protection.

transactions, like irrevocable transactions, cannot be aborted, but they also have the property that no other transaction can concurrently execute alongside them. Isolated transactions are useful for ensuring lock composition.

5.4.3 The Necessity of Full Invalidation

When irrevocable transactions execute in TMs, other non-irrevocable transactions, or revocable transactions, can execute along side irrevocable transactions as long as the commit of these revocable transactions do not cause irrevocable transactions to abort. To determine this, revocable transactions must verify that their commit does not prevent an active, irrevocable transaction from committing. To the best of our knowledge, all TMs that support irrevocable transactions use invalidation to ensure revocable transactions do not conflict with active irrevocable transactions.

Because our system uses full invalidation, there is no change in its semantics when irrevocable transactions are executed and only a minor amount of complexity is added to our design to support this type of behavior. Our system simply performs one addition check to verify that if a committing, revocable transaction conflicts with an active transaction that is irrevocable that the revocable transaction is aborted. This check is added just before handing the conflict to the CM. This prevents the CM from accidentally aborting the irrevocable transaction.

However, other systems that do not use any form of invalidation have been changed to support some form of invalidation so they can support the concurrent execution of irrevocable and revocable transactions [90]. If these TMs are not changed to use invalidation, they can still support isolated transactions or revocable read-only transactions that run concurrently with irrevocable transactions, but are seemingly unable to run revocable writer transactions concurrently with irrevocable transactions. This is because in order for a revocable writer transaction to commit while an irrevocable transaction is active, it must verify that its write set does not prevent an active irrevocable transaction from committing. It seems the only way to perform this action is to use visible readers in conjunction with invalidation.

5.4.4 LiT Policies

As shown in Figure 5.6, we use a six threaded example to demonstrate the different LiT policies. Thread T_1 executes `tx1()`, T_2 executes `tx2()`, T_3 executes `tx3()`, T_4 executes `inc2()`, T_5 executes `inc3()` and T_6 executes `inc4()`. Threads T_1 (`tx1()`) and T_6 (`inc4()`) do not conflict with any other thread, indicated by the call to `no_conflict()`. Thread T_2 has a conflict with thread T_4 while thread T_3 has a conflict with thread T_5. We use the following staggered start time. T_1 starts, then T_2 starts, then T_3 starts, and so on until T_6 starts.

A visualization of Figure 5.6 is shown in Figure 5.5. The LiT threads are labeled based on the locks they acquire: thread `tx1` uses lock L_1, thread `tx2` uses lock L_2 and thread `tx3` uses lock L_3. We do the same for locking threads: thread `lock L2` uses lock L_2, thread `lock L3` uses lock L_3 and thread `lock L4` uses lock L_4. We have used this naming convention in an attempt to make it easier to describe conflicts between transactions and locks: any transaction function or locking function that have the same number, conflict with one another. For example, `tx2` conflicts with `lock L2`, `tx3` conflicts with `lock L3`, while `tx1` and `lock L4` do not conflict, which can be noted by analyzing their names and seeing their numbers do not match.

5.4.4.1 LiT Full Lock Protection and Atomic Serialization

LiT full lock protection does not require any knowledge of locking conflicts to execute correctly. It ensures serializability by disallowing the execution of any lock-based or transaction-based critical section while the LiT transaction is executing. LiT full lock protection assumes that other transactions may acquire locks within them and therefore disallows other transactions from executing until the LiT transaction is complete. LiT full lock protection is equivalent to Ziarek et al.'s atomic serialization [91]. LiT full lock protection's maximum concurrent throughput at a given moment in time is expressed as follows:

$$m(LiT_{fl}) = t_l \oplus L_n \oplus T_{nl}$$

```
1  //--------------------------
2  // Thread T1
3  //--------------------------
4  void tx1() { atomic(t) { no_conflict(); } }
5
6  //--------------------------
7  // Thread T2
8  //--------------------------
9  void tx2() {
10    atomic(t) {
11       t.lock_conflict(L2);
12       inc2();
13    } end_atom
14 }
15
16 //--------------------------
17 // Thread T3
18 //--------------------------
19 void tx3() {
20    atomic(t) {
21       t.lock_conflict(L3);
22       inc3();
23    } end_atom
24 }
25
26 //--------------------------
27 // Thread T4
28 //--------------------------
29 void inc2() {
30    lock(L2);
31    ++g2;
32    unlock(L2);
33 }
34
35 //--------------------------
36 // Thread T5
37 //--------------------------
38 void inc3() {
39    lock(L3);
40    ++g3;
41    unlock(L3);
42 }
43
44 //--------------------------
45 // Thread T6
46 //--------------------------
47 void inc4() {
48    lock(L4);
49    no_conflict();
50    unlock(L4);
51 }
```

Figure 5.6: Six Threaded LiT Example.

t_l is a single transaction that acquires a lock while t_l is executing. L_n is the maximum number of lock-based critical sections that do not conflict with one another. T_{nl} is the maximum number of transactions that can be executed which do not have locks inside of them and do not conflict with one another. LiT full lock protection can only support the execution of one of the following: a LiT transaction, non-conflicting locks or non-LiT transactions (as denoted by the exclusive-or $\oplus$).

5.4.4.2 LiT TM-Lock Protection

LiT TM-lock protection can sometimes improve the performance of LiT full lock protection because it uses programmer annotations, to help identify locks that may conflict with transactions. Because the programmer has identified locks that conflict with LiT transactions, those locks that do not conflict with the LiT transaction can be run concurrently, increasing overall system throughput. LiT TM-lock protection requires the programmer to specify which locks are obtained inside of transactions before any lock is acquired. Once a transaction obtains exclusive access to a lock inside of it, all of the locks listed as conflicting are prevented from being acquired by other threads until the LiT transaction completes. LiT TM-lock protection requires that LiT transactions to be run as an isolated transaction. This is because other active transactions might attempt to acquire a lock that could conflict with the LiT transaction. The maximum concurrent throughput of TM-lock protection at a specific instance in time while executing a LiT is:

$$m(LiT_{tm}) = (t_l + L_{nt}) \oplus (L_n + T_{nl})$$

t_l is a single transaction that acquires a lock while it is executing. L_{nt} is the maximum number of lock-based critical sections that do not conflict with one another and do not conflict with t_l. L_n is the maximum number of locks that do not conflict with each other, but do conflict with t_l. T_{nl} is the maximum number of transactions that can be executed which do not have locks inside of them, do not conflict with L_n, and do not conflict with one another. TM-lock protection is a theoretical improvement over LiT full lock protection (and therefore atomic serialization) because locks that do not conflict with a LiT can be executed alongside a LiT transaction, or other non-conflicting

transactions, as illustrated with Figure 5.5. Notice that in Figure 5.5 TM-lock protection identifies that `lock L4`, `tx2` and `tx3` do not conflict and can therefore be run concurrently. It also identifies the same behavior for `lock L4` and `tx2` and `tx3`.

5.4.4.3 LiT TX-Lock Protection

LiT TX-lock protection offers the highest potential concurrent throughput of the LiT policies, but requires explicit local knowledge of locking conflicts. TX-lock protection relaxes the requirement of LiT transaction execution, allowing LiT transactions to run as irrevocable transactions rather than as isolated transactions. This optimization allows other revocable transactions to be run concurrently alongside a LiT transaction.

With LiT TX-lock protection, the programmer specifies which locks are used within each transaction, requiring specific local knowledge of locks and their interaction with transactions. The TM system can then relax the requirement of running LiT transactions as isolated and instead run them as irrevocable. While only one irrevocable transaction can be run at a time, other revocable transactions can be run alongside an irrevocable transaction, improving potential transaction concurrency over TM-lock protection and full lock protection. The maximum concurrent throughput LiT TX-lock protection can achieve at any moment in time is expressed as follows:

$$m(LiT_{tx}) = (t_l + L_{nt} + T_r) \oplus (L_n + T_{nl})$$

All the variables of the LiT TX-lock protection equation are the same as the LiT TM-lock protection equation with the exception of T_r. T_r is the maximum number of revocable transactions that can be executed concurrently alongside t_l as long as these transactions do not conflict with the set of locks in L_{nt}. As shown in Figure 5.5, the actual conflict time between the transactions and locks is equal to the TX-lock protection policy-induced conflict time.

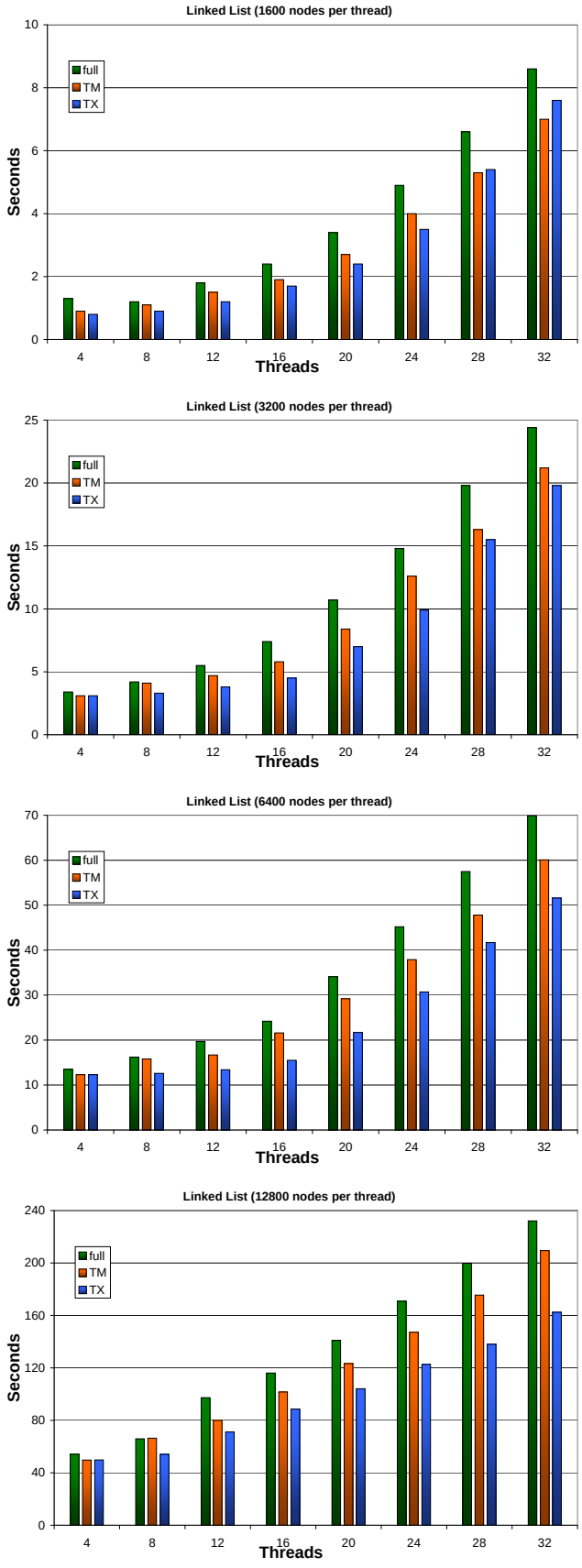


Figure 5.7: Linked List Benchmarks: LoT Lock Protection Policies.

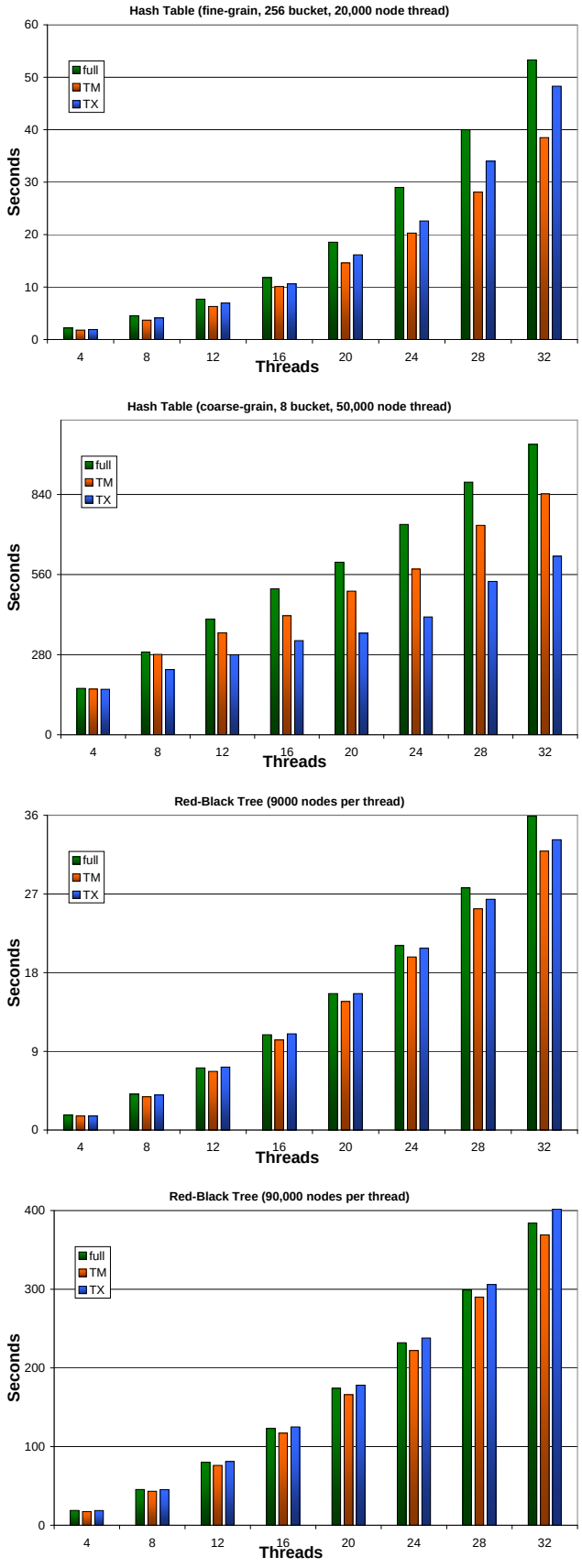


Figure 5.8: Hash Table and Red-Black Tree Benchmarks: LoT Lock Protection Policies.

5.5 Experimental Results

In this section we present the experimental results for our LATM. All benchmarks were performed on a 1.0 GHz Sun Fire T2000 supporting 32 concurrent hardware threads and 32 GB RAM. For all of the benchmarks with the exception of the red-black trees in Figure 5.10, the x-axis shows the number of active threads and the y-axis shows the total execution time in seconds. In Figure 5.10, the x-axis shows the number of inserts and lookups and the y-axis shows the total execution time in seconds. On the y-axis, the shorter the bar, the shorter the execution time. In other words, shorter bars represent more efficient executions in terms of performance.

LoT Policy Performance

Figures 5.7 and 5.8 display the execution time of the LoT policies on a 4-threaded through 32-threaded linked list, hash table, and red-black tree experimental model. Each benchmark was executed using full lock protection (left bar), TM-lock protection (middle bar), and TX-lock protection (right bar). The graphs show benchmarks starting from 4 threads, leading up to 32 threads. In the 4-threaded model, three separate containers are populated. Thread T_1 populates container C_1 with locks. Thread T_2 populates container C_2 with transactions. Container C_3 is concurrently populated by thread T_3 with locks, and thread T_4 with transactions.

LiT Policy Performance

Figures 5.9 and 5.10 display the execution time of the LiT policies on a 4-threaded through 32-threaded experimental model. With the exception of the red-black trees in Figure 5.10, we use the same basic 4-threaded model from the LoT experiments. The container population is identical to the LoT benchmarks, except that a LiT transaction is added which performs a lock-based `lookup()` followed by a lock-based `insert()`. For Figure 5.10, we focused on smaller threaded red-black tree experiments that used only 4-threaded and 8-threaded experiments. Instead of increasing the thread count, we doubled the tree size with each iteration. We believe these benchmarks provide valuable insight into the degrading performance of TM-lock protection.

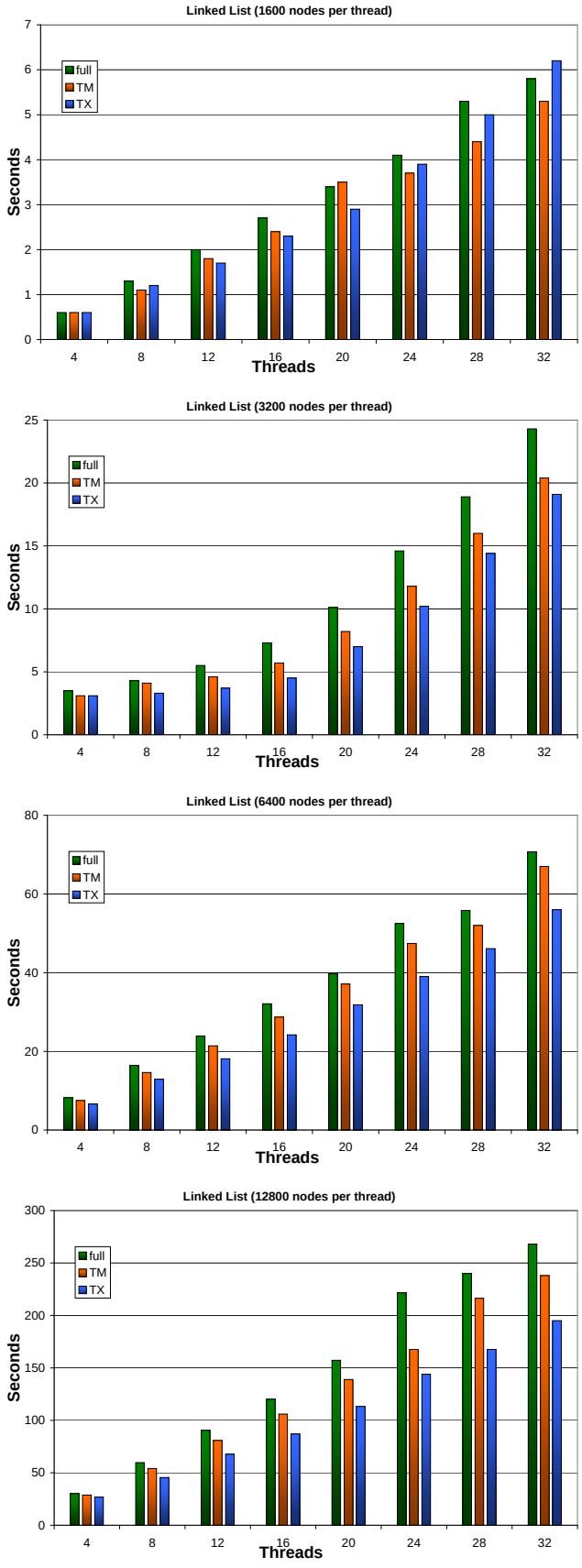


Figure 5.9: Linked List Benchmarks: LiT Lock Protection Policies.

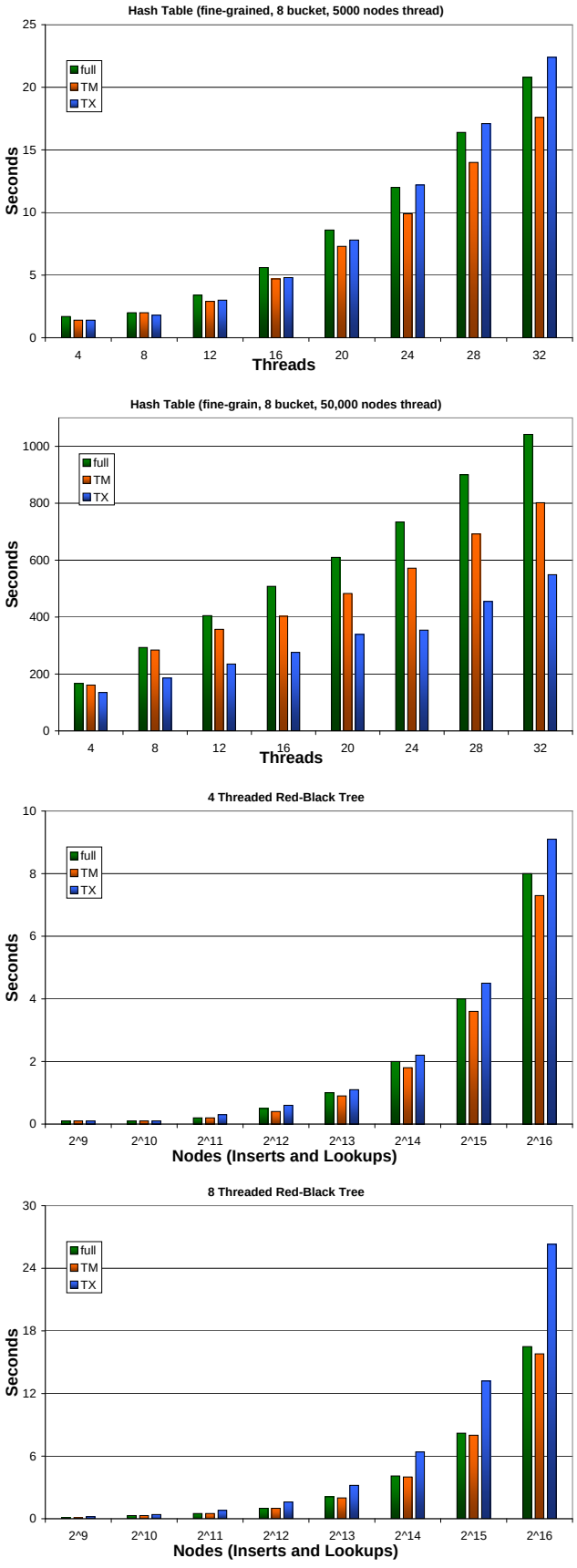


Figure 5.10: Hash Table and Red-Black Tree Benchmarks: LiT Lock Protection Policies.

5.5.1 Performance Summary

Our experimental results are surprising. TM-lock protection is consistently more efficient, in terms of total execution time, than full lock protection. However, TX-lock protection sometimes performs almost $\approx 2x$ slower than both TM-lock protection and full lock protection. The cumulative results of our experiments seem to indicate that TM-lock protection is a better candidate for practical software applications than TX-lock protection because (1) it has minimal programmatic overhead (e.g., each of our benchmarks only required one extra line of code for TM-lock protection) and (2) it is consistently more efficient than the current state-of-the-art for all of our experimental benchmarks.

These results are not intuitive. The mathematical analysis presented in Sections 5.3 and 5.4 seemed to illustrate that TX-lock protection should outperform full and TM-lock protection because TX-lock protection supports a greater amount of potential concurrent throughput. However, our experimental results tell a different story. That is, the execution time to support TX-lock's current algorithm can sometimes exceed that of the increased concurrency it gains. Therefore, even though TX-lock protection does indeed support a greater amount of concurrent throughput, the time it takes to achieve such behavior outweighs the benefit of the added concurrency.

Several factors contribute to the performance degradation of TX-lock protection compared to full and TM-lock protection. The two factors that we believe are among the most important are listed below. First, in order for TX-lock protection to produce a performance benefit over the other policies, the critical sections that TX-lock protection executes should have greater overhead than its algorithm's execution. Second, TX-lock protection must identify a number of locations where concurrent throughput can be increased, which would be missed by full or TM-lock protection. Otherwise, TX-lock protection will not produce enough extra concurrency to offset its algorithmic overhead and perform worse than if it did no conflict management at all.

As can be seen in the 4-threaded red-black tree LiT benchmark results (Figure 5.10), the performance divide between TX-lock protection and the other policies increases as the number of

operations performed on the red-black trees increases. This demonstrates that the time it takes to execute the critical section of the benchmark is less than the time it takes to execute the TX-lock protection algorithm. Otherwise, the performance difference between TX-lock protection and the other policies would remain static as the number of operations performed on the red-black tree increased. Comparing the 4-threaded and 8-threaded red-black tree LiT benchmarks to each other (again, Figure 5.10), one can observe a greater performance degradation of TX-lock protection in the 8-threaded red-black tree than is seen for the same workloads in the 4-threaded red-black tree. This illustrates that the algorithmic overhead of the TX-lock protection is greater than the extra concurrency it identifies that the other policies miss.

5.5.2 Performance Conclusion

TX-lock protection performs well compared to the other policies when the critical sections that are being executed take longer to execute than TX-lock protection's algorithm and such critical sections increase in their execution time when performed iteratively, such as performing iterative tail-inserts on a linked list (see Figures 5.7 and 5.9). In this way, if more threads are added to the workload, TX-lock protection may initially perform poorly compared to the other policies, but may eventually outperform them once the critical section execution time reaches some threshold.

However, if the critical section workload of the program remains constant (or nearly constant) as the number of threads increases for the workload, such as performing inserts on a red-black tree, TX-lock protection's performance will degrade at some rate proportional to the increasing number of threads (see the top-left hash table and red-black trees of Figures 5.8 and 5.10). While our experimental results show that TX-lock protection can perform upward of $\approx 2x$ compared to full lock protection (see the top-right benchmark in Figure 5.10), these results may be unreliable because its performance improvements are proportional to the application's critical section execution time. Because of this, in other cases TX-lock protection performs $\approx 2x$ worse than the other policies (see the bottom-right red-black tree of Figure 5.10).

On the other hand, our experiments demonstrate that TM-lock protection performs bet-

ter than full lock protection (i.e. TxLocks and atomic serialization) without regard to thread count, critical section execution time, data set size, or benchmark asymptotic complexity (see Figures 5.7, 5.8, 5.9 and 5.10). Furthermore, TM-lock protection requires less programmer effort than TX-lock protection making it easier for programmers to adopt. Due to these factors, our initial results indicate that TM-lock protection is probably more practical than TX-lock protection. At least this is the case for the benchmarks we have studied in this chapter. However, it is certainly possible that new benchmarks will result in different performance results that may require us to re-evaluate this conclusion.

5.6 The Future of Contention Management

The introduction of LATMs presents a new problem to be addressed by contention management (CM). In particular, CMs will likely be required to manage the forward progress of both locks and transactions alike. While an existing body of research has already been dedicated to the exploration of contention management (CM) for TM [35, 36, 79, 83], these efforts have been restricted to the investigation of transactional forward progress.

5.6.1 Unified Contention Management

Unified contention management (or UCM), as we call it, is a contention management system that controls the forward progress of both optimistic and pessimistic critical sections. Existing CMs, which control the forward progress of transactions, were originally built to manage critical sections that support failure atomicity. Unfortunately, locks are not failure atomic (see Section 5.4.2). Because of this, contention managers must manage the forward progress for locks in a different manner than the forward progress of transactions. In particular, the only point in a pessimistic critical section where a contention manager can manage a lock is just before its critical section begins as demonstrated in Figure 5.11. Due to this, existing CM policies cannot arbitrate lock-based forward progress in the same manner as they arbitrate transaction-based forward progress.

Instead, lock-based CMs must decide if a thread's lock-based critical section should be al-

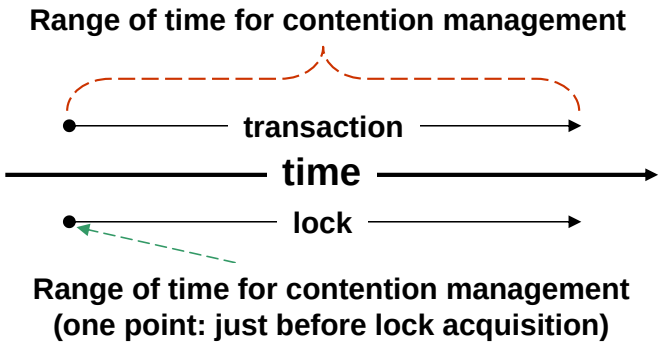


Figure 5.11: Differences in Transaction- and Lock-Based Contention Management.

lowed to make forward progress before any portion of the critical section is executed. This is a conceptual departure from CMs that only handle the forward progress guarantees of transactions. Such transaction-based CMs typically decide which transactions will be allowed to make forward progress after the transactions have executed in part or in full. The range of potential contention management for transactions is shown in Figure 5.11.

5.6.2 Our LATM UCM

In building our LATM, we began to explore the space of unified contention management by constructing a basic UCM. Because we have just begun to explore the UCM space, we currently only implement a single UCM policy for locks and transactions, which we call *UnifiedFair*, but we plan to significantly extend this work in the future.

Lock-based Forward Progress. UnifiedFair stalls the forward progress of locks for some constant time C if conflicting transactions are active. Once C has expired, UnifiedFair aborts all active transactions that are revocable and of equal or lesser priority than the lock-based critical section. In the event an irrevocable or isolated transaction is in-flight, UnifiedFair adds the lock-based critical section to its management queue which forces future irrevocable and isolated transactions to execute after the waiting lock-based critical section. This behavior prevents the lock's critical section from starving.

Transaction-based Forward Progress. Managing transaction-based forward progress in a UCM requires additional measures that are not present in CMs that only manage the forward progress of transactions. One problem of particular importance is that UCMs can indefinitely stall transactions as lock-based critical sections continue to take precedence over transactions, ultimately causing some transactions to starve. To prevent this and other malign behaviors, UnifiedFair has the following specification.

First, when a lock-based critical section is executing, all conflicting transactions are stalled until the already active lock-based critical section has completed. UnifiedFair then allows the stalled transactions to begin their execution and prevents new lock-based critical sections from interference with their execution. Unfortunately, this alone is insufficient. Because transactions can abort one another, transactions that were previously stalled by a lock-based critical section can be aborted by a committing transaction that has higher priority. This cycle could repeat indefinitely unless some mechanism is put in place to prevent it.

UnifiedFair prevents the above cycle from repeating indefinitely by raising a transaction's priority by the number of lock-based critical sections that have stalled it and the number of transactions that have aborted it. Eventually, using this dynamic priority assignment, similar to the one described in Chapter 4, no transaction will be indefinitely stalled.

5.6.3 Open Questions of UCM

While we believe our initial work on UnifiedFair is an important first step in UCM, there are many important problems that we have not yet addressed. Some, but not all, of those open problems are listed below:

- Transactional operations are inherently more costly than those operations inside of a lock-based critical section. This is because transactions must be failure atomic and therefore must perform some logging to unwind operations of aborted transactions or redo operations of a committed transaction. Operations inside a lock-based critical section do not perform

such operations so they are inherently cheaper to perform. Due to this, can we construct an operational weight formula that calculates the weight of lock-based operations and transactional operations, while taking into account how many times a transaction has been aborted and how long a lock-based critical section has been stalled?

- Once a lock-based critical section begins execution it generally must execute to completion. As such, is there a metric, other than the previously discussed stall period, in which the UCM can effectively arbitrate forward progress of lock-based critical sections?

- Is using the period of time it takes for a lock-based critical section to execute a reasonable metric for determining future execution behavior for the same lock-based critical section? If so, how should this value be used compared to transaction executions? If not, is it practical to require additional annotations on locking APIs regarding which methods they are being acquired and released from to help more accurately estimate future locking behavior (e.g., `lock(lockA, methodB)`)?

- In our UCM, a waiting lock with the same priority of an in-flight transaction can abort the transaction, yet a waiting transaction cannot abort an in-flight lock even if the lock is of lesser priority. What mechanisms should be put in place to prevent transactions from becoming second-class citizens to locks?

5.7 Conclusion

This chapter presented our unique LATM which allows programmers to use transactions and locks within the same program. We presented two new LATM policies, a coarse-grained policy and a fine-grained policy. Our coarse-grained policy, TM-lock protection, requires the programmer to add only one line of code to his or her system per lock that may conflict with a transaction, and can yield up to $\approx 1.4x$ improved efficiency over prior state-of-the-art systems. Our fine-grained policy, TX-lock protection, requires more programmer effort, but achieves up to $\approx 2x$ improved performance over the state-of-the-art systems for select benchmarks.

Our experimental results were surprising. TM-lock protection, our coarse-grained LATM policy, outperformed full lock protection for all of our experimental benchmarks, while TX-lock protection executed $\approx 2x$ slower than the other policies in some cases. Furthermore, TM-lock protection requires less programmer effort than TX-lock protection making it easier for programmers to adopt. Due to these factors, our initial results indicate that TM-lock protection is probably more practical than TX-lock protection. At least this is the case for the benchmarks we have studied in this chapter. However, it is certainly possible that new benchmarks will result in different performance results that may require us to revisit our initial conclusion.

We concluded with the introduction of a new area of research in contention management, what we call unified contention management (UCM). The introduction of LATMs will likely require that CMs manage the forward progress of transaction-based and lock-based critical sections. We explained why existing CMs may be unable to support the forward progress of lock-based critical sections and the basic design of UnifiedFair, our first attempt at a UCM. We closed by presenting several open UCM questions.

Chapter 6

Conclusion

In this work we presented a conflict detection strategy called full invalidation where a committing transaction resolves all conflicts that exist between it and other active transactions before it commits. Full invalidation has a number of advantages over validation such as improved performance, enforceable execution guarantees, reduced conflict speculation, bulk conflict analysis, and simplified integration of optimistic and pessimistic concurrency control. In this work we discussed the above topics with regard to invalidation. We also demonstrated areas where invalidation performs well compared to other conflict detection mechanisms and where invalidation does not perform so well. In the end, invalidation is like any other algorithm; it can perform more efficiently than other algorithms in the right conditions and it can perform less efficiently in the wrong conditions. As we have tried to demonstrate in this work, while we believe invalidation is powerful, it is not a panacea.

One of the most notable computational drawbacks of full invalidation is that it seems to require that all transactions' read sets be visible. As noted in several chapters of this work, visible read sets incur some degree of computational overhead not found when using invisible read sets. The benefit of visible read sets, of course, is that they can sometimes improve transaction throughput over what is possible using invisible read sets. As such, we spent a portion of this work (Chapters 2 and 4) exploring the ways to minimize the negative performance impact of visible read sets. While we have presented some novel ways in which the overhead of visible readers can be mitigated, we believe that a great deal more work should be dedicated to continuing to explore the space of

optimizing visible readers.

6.1 Optimizing TM for Contending Workloads and Memory-Intensive Transactions

Although invalidation has performance drawbacks it can also yield performance benefits. One such benefit is that invalidation can increase transaction throughput for contending workloads. This can be achieved by the contention manager of a fully invalidating TM because the contention manager has a complete picture of the transactions that must abort when a transaction commits. Because the contention manager knows which transactions will be aborted if a transaction commits, the contention manager can make informed decision that will likely result in increased transaction throughput over what is possible when using a validating TM for the same scenario. When this type of informed decision making is used in a system where there is transaction contention, transaction throughput can sometimes be increased by notable proportions. In Chapter 2, we presented concrete examples demonstrating which scenarios can benefit from this type of decision making.

Another benefit to using full invalidation is that memory-intensive transactions, those transactions that access many memory elements during their lifetime, may require notably fewer operations to perform opacity and conflict detection checks. Some results of these types of transactions were included in Chapter 2.

6.2 The Theory of Full Invalidation

Next, we provided a theoretical treatment to full invalidation. Our theoretical model used I/O automata theory from Lynch et al. [57] and our proof of correctness was based on conflict and view serializability from Papadimitriou [68].

In essence, our goal was to show that our full invalidation model satisfied the correctness criteria prescribed by conflict and view serializability [68]. Our proof required a new type of conflict graph because Papadimitriou's original conflict graph assumed a direct update system, yet, to maximize the benefits found in full invalidation the system should use deferred update. To

accommodate this, we defined a new type of conflict graph that we call a lazy conflict graph, which uses reads and writes based on deferred update. We then showed, as does Papadimitriou in the original work, that all of the concurrent histories accepted by our full invalidation model construct acyclic lazy conflict graphs.

6.3 User-Defined Priority-Based Transactions

We also demonstrated that real-time TM systems that might require user-defined priority-based transactions can achieve upwards of a 100× throughput improvement over validation when using invalidation. While user-defined priority-based transactions can be respected using validation, such systems seem to notably decrease transaction throughput. This is because a validating system does not have visible readers. Because a validating TM does not have visible readers, and therefore cannot detect conflicts between active transactions, another mechanism must be found in order to respect priority-based transactions. The mechanism we used to respect transactional priority was the following. When a transaction commits, it scans all other active transactions' priority. If any active transaction has a higher priority than the committing transaction, the committing transaction is aborted. This mechanism ensures transactional priority is respected in a validating TM. The downside of this approach is that transactions that may not have conflicts are aborted because their priority is lower than another active transaction.

6.4 Lock-Aware Transactional Memory and Unified Contention Management

Lock-aware transactional memory (LATM) allows transactional memory to execute transaction-based critical sections along with lock-based critical sections. LATM can also allow for locks to be acquired inside the body of a transaction, but doing so can require a change to the semantics of the transaction's critical section. On the other hand, LATM can allow for transactions to be executed inside the body of a lock's critical section, although the semantics of the lock and the transaction in this scenario may not necessarily require a change.

In addition to presenting interesting theoretical questions, LATM has a practical level of

importance associated with it. Without LATM transactions may violate the mutual exclusion property that is necessary for the correct behavior of lock-based critical sections. If LATM is not supported and a programmer wishes to integrate transactions into legacy code that has locks within it, it may be necessary to rewrite the legacy software so it only uses transactions (a potentially unrealistic requirement).

In this work, we showed that invalidation may be necessary for transactions and locks to achieve the highest possible degree of transaction and lock throughput. The reason for this is perhaps best demonstrated using a counterexample. Consider a TM that only uses validation and is being changed to become a lock-aware TM and how it should handle the following scenario. Consider the scenario where an active transaction needs to acquire a lock within its transactional body (a lock inside of a transaction, or LiT), and once the lock has been acquired the semantics of the transaction have changed such that the transaction can no longer be aborted. In other words, the LiT transaction becomes irrevocable. How can the TM guarantee that other active transactions will not commit in such a way that will require the LiT to abort?

In a validating TM, a transaction is only allowed to commit after it has verified that its read and write data does not conflict with any of the previously committed transactions. As such, in a validating TM if an irrevocable LiT is active and another transaction commits, such a commit may cause the irrevocable LiT to abort; a violation of the irrevocable property of the transaction. Therefore, a validating TM seems only capable of supporting the execution of one LiT at a time without the concurrent execution of any other transactions. Other transactions cannot execute concurrently with the LiT because they may inadvertently cause the irrevocable LiT to abort. In an invalidating TM, because transactions identify the conflicts they have with other transactions before they commit, any number of revocable transactions can execute concurrently with the irrevocable LiT. When the revocable transactions reach their commit phase, they must verify they do not have a conflict with the irrevocable LiT. If the revocable transactions do have a conflict with the LiT, they must self-abort. By allowing revocable transactions to concurrently execute alongside an irrevocable LiT the overall transaction and lock throughput can be notably

increased as demonstrated in Chapter 5.

Bibliography

[1] Dual core era begins, PC makers start selling Intel-based PCs: Intel dual-core processor-powered PC systems first to market. Technical report, Intel Corporation, April 2005.

[2] The first step in the multi-core revolution. Technical report, Intel Corporation, April 2005.

[3] Ali-Reza Adl-Tabatabai, David Dice, Maurice Herlihy, Nir Shavit, Christos Kozyrakis, Christoph von Praun, and Michael Scott. Potential show-stoppers for transactional synchronization. In PPOPP, page 55, 2007.

[4] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. ACM Queue, 4(10):24–33, 2006.

[5] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Transactional programming in a multi-core environment. In Katherine A. Yelick and John M. Mellor-Crummey, editors, PPoPP, page 272. ACM, 2007.

[6] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. volume 26, pages 59–69. IEEE Computer Society Press, Los Alamitos, CA, USA, 2006.

[7] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst., 1(1):6–16, 1990.

[8] Joshua Bloch. Effective Java (2nd Edition) (The Java Series). Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[9] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422–426, 1970.

[10] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. SIGPLAN Not., 43(6):68–78, 2008.

[11] Shekhar Borkar, Pradeep Dubey, Kevin Kahn, David Kuck, Hans Mulder, Steve Pawlowski, and Justin Rattner. Platform 2015: Intel processor and platform evolution for the next decade. Technical report, Intel Corporation, March 2005.

[12] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? Queue, 6(5):46–58, 2008.

[13] David Culler, Jaswinder Pal Singh, and Anoop Gupta. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers, 1999.

[14] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In PPoPP '10: Proc. 15th ACM Symp. on Principles and Practice of Parallel Programming, Jan 2010.

[15] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, pages 157–168, New York, NY, USA, 2009. ACM.

[16] David Dice, Mark Moir, and Nir Shavit. Sun Microsystems: Transactional memory.

[17] David Dice, Ori Shalev, and N. Shavit. Transactional locking II. In Proc. of the 20th International Symposium on Distributed Computing (DISC 2006), pages 194–208, 2006.

[18] David Dice and Nir Shavit. What really makes transactions faster? ACM SIGPLAN Workshop on Transactional Computing, June 2006.

[19] David Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In Proc. of the 2007 International Symposium on Code Generation and Optimization (CGO), 2007.

[20] E. W. Dijkstra. Solution of a problem in concurrent programming control. Commun. ACM, 8(9):569, 1965.

[21] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[22] Keir Fraser and Tim Harris. Concurrent programming without locks. ACM Trans. Comput. Syst, 25(2), 2007.

[23] Hector Garcia-Molina and Gio Wiederhold. Read-only transactions in a distributed database. ACM Trans. Database Syst., 7(2):209–234, 1982.

[24] Justin Gottschlich. Exploration of lock-based software transactional memory. Master's thesis, University of Colorado at Boulder, 2007.

[25] Justin E. Gottschlich, Dan A. Connors, Dwight Y. Winkler, Jeremy G. Siek, and Manish Vachharajani. Lock-aware transactional memory. In ASPLOS '09: Proc. of the fourteenth international conference on architectural support for programming languages and operating systems (poster), March 2009.

[26] Justin E. Gottschlich and Daniel A. Connors. DracoSTM: A practical C++ approach to software transactional memory. In ACM SIGPLAN Library-Centric Software Design (LCSD), 2007.

[27] Justin E. Gottschlich and Daniel A. Connors. Extending contention managers for user-defined priority-based transactions. In Proceedings of the 2008 Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods, April 2008.

[28] Justin E. Gottschlich and Daniel A. Connors. Optimizing consistency checking for memory-intensive transactions. In ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 2008.

[29] Justin E. Gottschlich, Jeremy G. Siek, Paul J. Rogers, and Manish Vachharajani. Toward simplified parallel support in C++. In Proceedings of the Fourth International Conference on Boost Libraries (BoostCon). May 2009.

[30] Justin E. Gottschlich, Jeremy G. Siek, and Manish Vachharajani. Proving conflict serializability for full invalidation. In Proceedings of the Second Workshop on the Theory of Transactional Memory, September 2010.

[31] Justin E. Gottschlich, Jeremy G. Siek, Manish Vachharajani, Dwight Y. Winkler, and Daniel A. Connors. An efficient lock-aware transactional memory implementation. In Proceedings of the Fourth International ACM Workshop on ICOOOLPS. In conjunction with ECOOP. July 2009.

[32] Justin E. Gottschlich, Manish Vachharajani, and Jeremy Siek G. An efficient software transactional memory using commit-time invalidation. In Proceedings of the International Symposium on Code Generation and Optimization (CGO), April 2010.

[33] Justin E. Gottschlich, Manish Vachharajani, and Jeremy Siek G. An efficient software transactional memory using commit-time invalidation. In Proceedings of the International Symposium on Code Generation and Optimization (CGO), April 2010.

[34] Jim Gray and Andreas Reuter. Transaction Processing : Concepts and Techniques (Morgan Kaufmann Series in Data Management Systems). Morgan Kaufmann, October 1992.

[35] Guerraoui, Herlihy, and Pochon. Polymorphic contention management. In DISC. LNCS, 2005.

[36] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In Marcos Kawazoe Aguilera and James Aspnes, editors, PODC, pages 258–264. ACM, 2005.

[37] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In Proceedings of the ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 175–184, New York, NY, USA, 2008. ACM.

[38] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. ACM Comput. Surv., 15(4):287–317, 1983.

[39] Tim Harris and Keir Fraser. Language support for lightweight transactions. In OOPSLA, pages 388–402, New York, NY, USA, 2003. ACM.

[40] Tim Harris, James R. Larus, and Ravi Rajwar. Transactional Memory, Second Edition. Morgan and Claypool, 2010.

[41] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, PPoPP, pages 48–60. ACM, 2005.

[42] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. ACM SIG Notices, 41(6):14–25, June 2006.

[43] Armin Heindl and Gilles Pokam. Modeling software transactional memory with anylogic. In Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques, pages 1–10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[44] John Hennessy and David Patterson. Computer Architecture - A Quantitative Approach. Morgan Kaufmann, 2003.

[45] Maurice Herlihy. Wait-free synchronization. ACM Trans. Program. Lang. Syst., 13(1):124–149, 1991.

[46] Maurice Herlihy. A methodology for implementing highly concurrent data objects. ACM Transactions on Programming Languages and Systems, 15(5):745–770, November 1993.

[47] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In PPoPP. ACM, 2008.

[48] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 253–262, New York, NY, USA, 2006. ACM.

[49] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In Proceedings of the symposium on principles of distributed computing, pages 92–101, New York, NY, USA, 2003. ACM.

[50] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In Proceedings of the International Symposium on Computer Architecture. May 1993.

[51] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Elsevier, Inc., 2008.

[52] C. A. R. Hoare. Monitors: an operating system structuring concept. Commun. ACM, 17(10):549–557, 1974.

[53] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21:558–565, July 1978.

[54] James R. Larus and Ravi Rajwar. Transactional Memory. Morgan and Claypool, 2006.

[55] Doug Lea. Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[56] Liu and Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. JACM: Journal of the ACM, 20, 1973.

[57] Nancy A. Lynch, Michael Merritt, William E. Weihl, and Alan Fekete. Atomic Transactions: In Concurrent and Distributed Systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[58] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. revised, University of Rochester, Computer Science Department, May 2006.

[59] Virendra Jayant Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 227–236, New York, NY, USA, 2008. ACM.

[60] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: log-based transactional memory. In High-Performance Computer Architecture, 2006. The Twelfth International Symposium on, pages 254–265, 2006.

[61] J. Eliot B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. PhD thesis, 1981.

[62] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. Sci. Comput. Program., 63(2), 2006.

[63] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, pages 365–375, Washington, DC, USA, 2007. IEEE Computer Society.

[64] Kunle Olukotun, Lance Hammond, and James Laudon. Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency. Morgan and Claypool, 2007.

[65] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Michael Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2BSD file system. Technical report, Berkeley, CA, USA, 1985.

[66] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. ACM Trans. Program. Lang. Syst., 4(3):455–495, 1982.

[67] C. H. Papadimitriou. Serializability of concurrent data base updates. Technical report, Cambridge, MA, USA, 1979.

[68] Christos Papadimitriou. The theory of database concurrency control. Computer Science Press, Inc., New York, NY, USA, 1986.

[69] Insung Park, Michael Voss, Seon Wook Kim, and Rudolf Eigenmann. Parallel programming environment for openMP. Scientific Programming, 9(2-3):143–161, 2001.

[70] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. Java Concurrency in Practice. Addison-Wesley Professional, 2005.

[71] Ravi Rajwar and Philip A. Bernstein. Atomic transactional execution in hardware: A new high performance abstraction for databases. In Workshop on High Performance Transaction Systems, 2003.

[72] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In MICRO, pages 294–305. ACM/IEEE, 2001.

[73] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In MICRO '08, 2008.

[74] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing conflicting transactions in an STM. SIGPLAN Not., 44(4):163–172, 2009.

[75] Parameswaran Ramanathan and Moncef Hamdaoui. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. IEEE Trans. Comput., 44(12):1443–1451, 1995.

[76] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In Thomas C. Bressoud and M. Frans Kaashoek, editors, SOSP, pages 87–102. ACM, 2007.

[77] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 47–56, New York, NY, USA, 2010. ACM.

[78] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Ben Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In PPOPP. ACM SIGPLAN 2006, March 2006.

[79] William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In Marcos Kawazoe Aguilera and James Aspnes, editors, PODC, pages 240–248. ACM, 2005.

[80] William N. Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In Workshop on Concurrency and Synchronization in Java Programs, July 2004.

[81] Michael L. Scott. Sequential specification of transactional memory semantics. In ACM SIGPLAN Workshop on Transactional Computing. Jun 2006. Held in conjunction with PLDI 2006.

[82] Nir Shavit and Dan Touitou. Software transactional memory. In Proceedings of the Principles of Distributed Computing. Aug 1995.

[83] Michael F. Spear, Luke Dalessandro, Virendra Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In PPoPP, February 2009.

[84] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In Proceedings of the 20th International Symposium on Distributed Computing, Sep 2006.

[85] Michael F. Spear, Maged M. Michael, and Michael L. Scott. Inevitability mechanisms for software transactional memory. In Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing. Feb 2008.

[86] Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: scalable transactions with a single atomic instruction. In SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, pages 275–284, New York, NY, USA, 2008. ACM.

[87] Michael F. Spear, Andrew Sveikauskas, and Michael L. Scott. Transactional memory retry mechanisms. In PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing, pages 453–453, New York, NY, USA, 2008. ACM.

[88] Werner Vogels. File system usage in Windows NT 4.0. In SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles, pages 93–109, New York, NY, USA, 1999. ACM.

[89] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In ACM SIGPLAN Workshop on Transactional Computing, February 2008.

[90] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In SPAA, 2008.

[91] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for Java. In ECOOP, pages 129–154, 2008.

[92] Craig Zilles and David Flint. Challenges to providing performance isolation in transactional memories. In Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking, pages 48–55, Jun 2005.

Appendix A

Appendix

A.1 Atomic Operations

An *atomic operation* is a set of instructions that are executed together as a single, indivisible operation. Atomic operations are supplied by the processor's instruction set architecture (ISA) and are the basis of all higher level process synchronization types, like locks, monitors, and transactions. Some common atomic operations are:

- test-and-set (TAS) – Reads a value from a specified address, compares it to the first supplied parameter and if the values match, sets the specified address value to the second supplied value and returns the new value. If the values do not match, retry the operation. TAS operations are the basis for mutual exclusion locks.

- test and test-and-set – Performs test-and-set, but only after an initial condition is passed, verifying the specified address matches the compared value. This instruction is used to reduce the memory contention inherent in TAS.

- compare-and-swap (CAS) – Reads a value from a specified address, compares it to a first supplied value and if the values match, sets the specified address value to a second supplied value. If the values do not match, the operation does not change the specified address. Returns a value to inform the client if the operation succeeded or failed. Non-blocking atomic primitives are sometimes implemented using CAS.

- load-link / store-conditional (LL/SC) – LL/SC is broken into two distinct parts: LL and SC. The LL portion of the operation reads and returns a value from a specified address. The SC portion of the operation stores a specified value in the address if no updates have been made to the address since it was read. Non-blocking atomic primitives are sometimes implemented using LL/SC. The first software TM system was built using LL/SC.

A.2 Pessimistic and Optimistic Critical Sections

Critical sections can be pessimistic or optimistic. Pessimistic critical sections limit their critical section execution to a single thread. Locks are an example of a synchronization mechanism that use pessimistic critical sections. Optimistic critical sections allow unlimited concurrent threaded execution. Transactions are an example of a synchronization mechanism that can use optimistic critical sections[1] .

A.2.1 Truly Optimistic Critical Sections

Truly optimistic critical sections are those critical sections which allow multiple *conflicting* critical sections to be executed simultaneously. A deferred update (or lazy acquire) TM system is an example of a truly optimistic critical section. A direct update (or eager acquire) TM system is not a truly optimistic critical section.

Truly optimistic critical sections are important because they allow simultaneous conflicting critical section execution, as opposed to disallowing such behavior. It is important to allow conflicting critical section execution because to do otherwise would prematurely restrict concurrently executing threads to pessimistic (serial) execution which could degrade performance.

To demonstrate this concretely, consider two transactions, called T_1 and T_2, executing the same critical section. Transaction T_1 starts first and tentatively writes to memory location M. Transaction T_2 then starts and tries to write to memory location M. In a truly optimistic TM

[1] Note that some TM designers revoke the need to implement them optimistically[**?**, **?**].

system, T_2 would be allowed to tentatively write to location M while T_1 is also writing to M. This behavior then allows T_2 to commit before T_1 in the event T_2 completes before T_1. In comparison, if the TM system is not truly optimistic, once T_1 writes to M, T_2 must stall until T_1 completes. This pessimistically degrades the performance of the system by prematurely deciding that T_1's transactional execution should have higher priority than T_2's.

A.3 Concurrent Objects

The principal data object used to communicate in a concurrent system is a *concurrent object* [45]. Concurrent objects, also known as shared data objects, are used across multiple processes in a concurrent system. These objects relay information about program state between concurrently executing processes. The synchronization mechanisms used to implement concurrent objects determine the guarantees provided by the overall concurrent system (e.g., wait-free, lock-free, obstruction-free, deadlock-free, etc.).

Concurrent objects can be implemented in a *blocking* or *non-blocking* fashion. Blocking systems use pessimistic critical sections at their core (e.g., mutual exclusion locks). Non-blocking systems use atomic primitives that avoid using pessimistic critical sections. The manner in which a TM system is built, blocking or non-blocking, determines a number of its visible characteristics (details to follow).

A.4 Synchronization Mechanisms

A synchronization mechanism is a tool used to coordinate access to concurrent objects. There are a number of synchronization mechanisms used in parallel programming. Some of the important synchronization mechanisms in relation to TM are:

- Barrier – A barrier is a programmer specified location in a program where all threads must reach before any single thread can make forward progress. Barriers can be used to guarantee thread fairness and can therefore naturally avoid classical parallel problems such

as thread starvation.

- Mutual Exclusion (mutex) Lock – A mutual exclusion lock creates a pessimistic critical section where only one thread can execute a region of code controlled by a lock variable. To enter the code region the lock variable is acquired. Once the code region is exited, the lock is released. Locks come in two forms: fine-grained locking and coarse-grained locking. Locks are the most commonly used synchronization mechanism and are usually implemented using the atomic primitive TAS.

- Monitor – A monitor is an extension of a lock. Monitors control access to several code regions that use the same concurrent objects. Only one thread can access any one of the guarded code regions. Monitors are usually re-entrant safe, meaning one thread accessing a protected code region can also access any of the other monitor protected code regions without deadlocking. Locks are the underlying implementation of monitors.

- Semaphore – A semaphore is a lock with an associated counter. Semaphores are useful for controlling access to a finite set of resources, like file handles or array elements. Unlike a lock, semaphores allow multiple threads to enter its critical section. The number of threads allowed to enter a semaphore's critical section is based on the number of resources that are available and is specific to a particular software or hardware environment. Counters associated with semaphores are usually initialized to the maximum number of resources available and are decremented when a thread enters its critical section. When the semaphore's counter reaches 0, all of its resources have been exhausted. Threads waiting to enter the semaphore's critical section must wait until one thread exits, releasing a resource. Semaphores can be implemented with CASes.

- Non-blocking Atomic Primitives – Non-blocking atomic primitives implement concurrent objects without blocking the forward progress of other threads. Non-blocking synchronization techniques are designed to avoid the use of mutual exclusion locks. Because of

this, non-blocking techniques are guaranteed to exhibit deadlock freedom (e.g., they cannot deadlock a software system) even in the event of thread failure. Non-blocking atomic primitives are usually implemented using compare-and-swap (CAS) and load-link / store-conditional (LL/SC).

A.5 Problems with Locks

Mutual exclusion locks are the prevalent form of synchronization used in parallel programs [51]. There are a number of reasons why locks have been popularized, we list two below. First, locks have near universal architectural support across processors, which makes them widely available for use. Second, programmers tend to find their initial learning curve to be low. The concept of a mutual exclusion lock is similar to the concept of a lock on a house or car door, so the idea is easy for most programmers to understand.

Unfortunately, locks have many shortcomings, many of which are not seen until they are used within a software program, extensively. Abundant use of locks in the same software system can lead to various problems. Some of the most common lock-based problems are listed, and described briefly, below.

- Deadlock – Deadlocks occur when two or more processes obtain a resource (e.g., a lock) that is required by the other to make forward progress. When the processes are unwilling to relinquish ownership of their obtained resource, the threads are unable to make forward progress.

- Livelock – Livelocks occur when forward progress is perceived to happen, but does not actually happen. Livelocks can occur for a variety of reasons, but are usually the result of the repeated partial execution of an operation.

  An example of livelock is a thread executing nine out of ten steps for one overall operation. Suppose that the tenth step cannot be performed due to some constraint. The thread then rolls back and starts the overall operation again at step one. Livelock occurs if this scenario

repeats itself indefinitely. In particular, if the tenth step cannot be successfully executed, the thread will never make true forward progress, although it seems as though the thread is performing useful work as steps one through nine are continually executed.

- Lock Convoy – Lock convoys are performance degradation events that arise when multiple processes fail to acquire the same lock and then sleep, relinquishing their scheduled quantum.

  An example of a lock convoy is as follows. Consider thread H as the holder of a lock L. A set of N waiting threads, called W_N, then try to acquire a lock, fail, and relinquish their scheduling quantum, putting each thread in the set to sleep. Thread H then releases lock L, performs more work which eventually requires lock L again and obtains lock L a second time, completing its scheduling quantum. The array of W_N threads then awake, try and fail to acquire lock L and go back to sleep. As long as this process repeats, all of the threads in the W_N set will stall indefinitely.

- Priority Inversion – Priority inversion occurs when a process of lower priority holds a resource that a process of higher priority requires. The higher priority process is then stalled from making forward progress until the lower priority process completes.

A.5.1 Blocking and Non-Blocking Transactions

Transactions can be implemented in a blocking or non-blocking manner. Both blocking and non-blocking advocates have presented compelling arguments for both sides. Below is a brief summary of some such arguments.

In general, a primary motivation for non-blocking TM systems is that non-blocking TMs are scalable; non-blocking systems, by definition, ensure forward progress under certain conditions. Blocking implementations may not easily provide similar guarantees. Non-blocking implementations can also continue to function under thread failure. In blocking implementations, if a thread

fails while holding a lock, the system may eventually deadlock. Lastly, most non-blocking implementations can avoid many of the problems that plague locks, such as deadlock, priority inversion, and lock convoy [46, 59]. However, avoiding these problems is specific to the non-blocking implementation; not all non-blocking systems avoid these issues.

Those in favor of lock-based (or blocking) TMs tend to show higher overall transaction throughput. There is a growing body of evidence demonstrating that blocking TM systems outperform non-blocking ones [14, 18, 17, 21, 26, 33, 86]. Because one of the most visible concerns of TM is its performance, building systems that demonstrate good performance may increase the potential for TMs to be adopted by the hardware and software communities as a practical solution to parallel programming. Furthermore, recent research has demonstrated that certain problems thought to be lock-specific, such as priority inversion, can occur in non-blocking TM systems and in some cases are more likely to occur due to the non-blocking system's underlying complexity and required non-blocking guarantees [27].